DOCTOR OF PHILOSOPHY

The GraphGrind Framework: Fast Graph Analytics on Large Shared-Memory Systems

Sun, Jiawen

Award date:
2018

Awarding institution:
Queen's University Belfast

The GraphGrind Framework: Fast Graph Analytics on Large Shared-Memory Systems

Jiawen Sun, PhD candidate

School of Electronics, Electrical Engineering and Computer Science

Queen's University of Belfast

A thesis submitted for the degree of

Doctor of Philosophy

April 17, 2018

This work is supported by the European Community's Seventh Framework Programmer (FP7/2007-2013) under the Adaptive Scalable Analytics Platform (ASAP) project, grant agreement no. 619706, and by the United Kindom EPSRC under grant agreement EP/L027402/1.

Abstract

As shared memory systems support terabyte-sized main memory, they provide an opportunity to perform efficient graph analytics on a single machine. Graph analytics is characterised by frequent synchronisation, which is addressed in part by shared memory systems. However, performance is limited by load imbalance and poor memory locality, which originate in the irregular structure of small-world graphs. This dissertation demonstrates how graph partitioning can be used to optimise (i) load balance, (ii) Non-Uniform Memory Access (NUMA) locality and (iii) temporal locality of graph partitioning in shared memory systems. The developed techniques are implemented in GraphGrind, a new shared memory graph analytics framework.

At first, this dissertation shows that heuristic edge-balanced partitioning results in an imbalance in the number of vertices per partition. Thus, load imbalance exists between partitions, either for loops iterating over vertices, or for loops iterating over edges. To address this issue, this dissertation introduces a classification of algorithms to distinguish whether they algorithmically benefit from edge-balanced or vertex-balanced partitioning. This classification supports the adaptation of partitions to the characteristics of graph algorithms. Evaluation in GraphGrind, shows that this outperforms state-of-the-art

graph analytics frameworks for shared memory including Ligra by 1.46x on average, and Polymer by 1.16x on average, using a variety of graph algorithms and datasets.

Secondly, this dissertation demonstrates that increasing the number of graph partitions is effective to improve temporal locality due to smaller working sets. However, the increasing number of partitions results in vertex replication in some graph data structures. This dissertation resorts to using a graph layout that is immune to vertex replication and an automatic graph traversal algorithm that extends the previously established graph traversal heuristics to a 3-way graph layout choice is designed. This new algorithm furthermore depends upon the classification of graph algorithms introduced in the first part of the work. These techniques achieve an average speedup of 1.79x over Ligra and 1.42x over Polymer.

Finally, this dissertation presents a graph ordering algorithm to challenge the widely accepted heuristic to balance the number of edges per partition and minimise edge or vertex cut. This algorithm balances the number of edges per partition as well as the number of unique destinations of those edges. It balances edges and vertices for graphs with a power-law degree distribution. Moreover, this dissertation shows that the performance of graph ordering depends upon the characteristics of graph analytics frameworks, such as NUMA-awareness. This graph ordering algorithm achieves an average speedup of 1.87x over Ligra and 1.51x over Polymer.

Acknowledgements

There are so many people to support my PhD study and this thesis, and I would like to thank all of them.

First, I am fully thankful for my supervisor Hans Vandierendonck and Dimitrios S.Nikolopoulos for the advice and inspiration that they gave me during my studies. They introduced me parallel computing, graph analytics, also the English writing skills, and gave me a lot of useful suggestions for my research and future. I am very lucky to meet them when I start my PhD study in the Queen's University Belfast(QUB) as a Chinese student. Especially, in the beginning, Hans spent a lot of time to answer my questions every week to help me have progress. He also helped me improve my code, writing skills and my English. They gave me a lot of help and supports for research and paper writing. They taught me the real meaning of research, the whole process is tough and hard, but I learned a lot of things. For instance, do not give up and lose faith, keep going on, you will get some fruits.

I am also appreciate to all the people in HPDC cluster of QUB, they gave me a lot of useful feedback during my PhD study. They helped me with my conference presentation, and gave me many useful feedback for my research.

During my PhD study, my family gave me a lot of supports. My grandmother, Yuying Liang, inspired me to have a PhD study at first. My mother, Yan Liang encouraged me and stood by my side when I was down these years. My aunt, Hong Liang always supported my ideas and behaviors. Because of their supports, I could finish writing this thesis. Meanwhile, I want to thank my brother Jin Xie, and my cousin Chenyu Zhang. They were to accomplish my mother, motivated me to pursue my dreams these years. Without them, I would not have come this so far. I love them so much, no matter what happens, we are still a family.

Finally, I am fully thankful for my father, Huadong Xie, who was taught me "Today is better than yesterday, that is hope.". He was a very kind-hearted man. He taught me that a smile is the most charming part as a person forever. His words always encourage me when I am depression. I love you, dad. Rest in peace.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many problems in social network analysis, data mining and machine learning can be solved using graph-structured analysis [61, 62, 109]. Graph analytics is an important and computationally demanding class of data analytics. Graphs represent data as relationships between vertices, which drawn from real-world data sets, such as social network, internet and commercial services. These graphs are highly clustered, which results in low diameter (small-world) [97]. They usually follow skewed degree distribution that the number of edges of vertices exhibits a power-law distribution(scale-free) [7, 101]. Graph analytics specifies complex parallel computations in a simple way using relationship graphs.

In order to achieve efficient graph analytics, state-of-the-art uses a programming model for parallel graph computation. The model uses graph algorithms to compute values of vertices of a graph iteratively. When it computes a value of a vertex, it needs data from its adjacent vertices or incoming edges as input, and the computation value is communicated along outgoing edges [65, 91]. The computation stops until the values of vertices are converged, or a fixed number

of iterations have been performed [59, 63, 65, 73].

Hence, graph analytics is a data-driven [19, 40, 44] computation process. It performs a large number of computations on graph data to explore the structure of a graph. Graphs of large-scale graph analytics have several characteristics [5, 61]: (i) large volume, (ii) sparse connectivity, and (iii) irregular relationships between vertices. Hence, it is non-trivial to get efficient graph analytics due to these specific characteristics. The following properties present some significant challenges for efficient graph analytics.

- **Load imbalance.** The graph analytics models need to update computation values following incoming or outgoing edges, the work per vertex is not balanced due to the irregularity. It is difficult to parallel a graph analytics problem due to the imbalance computation loads.

- **Poor memory locality.** During each computing iteration, graph analytics requires to read and/or write values associated to the end-points of edges, which may be randomly dispersed through memory depending on the structure of the graph. Memory locality of graph analytics is thus highly data-dependent [61]. The iterative computations do not have too much locality due to the irregularly.

- **Memory-bound.** Graph analytics is a cache inefficient process due to poor memory locality. Since small-world graphs are sparse connected, it is hard to predict and optimise the data access patterns. Also, the large volume of graph data requires high data access rate. The execution time of graph analytics will be controlled by the latency of memory accesses.

State-of-the-art programming models try to speedup graph analytics by dealing with these challenges. In general, these models focus on three computing

systems: **(i) distributed memory systems, (ii) disk-based systems** and **(iii) shared memory systems**.

Distributed memory systems have multiple processors where each processor has its own private memory. They are able to scale to graphs by distributing graph partitions across distributed nodes [63]. Distributed memory systems can deal with the increasing amount of graph data by adding simple nodes [63, 73]. However, it is complicated to find efficient graph partitioning methods to minimise communication and balance computation between nodes for power-law graphs [33, 56].

Disk-based systems can do efficient graph analytics on graphs with billions of edges on a personal computer [54,96] without requirement of accessing a cluster. However, the scalability of graph analytics on a personal computer is limited by the increasing volume of graph data. The scalability of parallelism is also limited by the hardware limitation of a personal computer.

Shared memory systems can process hundred billion edges with fast execution time, such as Ligra [85]. Recently, several state-of-the-art graph analytics models [38, 54, 72, 80, 85–87, 102, 108] have proven that shared graph analytics can be fast, easy and scalable. Moreover, the development of multi-core results in a rapid increase in CPU memory size and core counts, which provides an opportunity to apply efficient graph analytics in a single machine [108].

There is increasing evidence [38, 54, 72, 80, 102, 108] that shared memory machines with terabyte-sized main memory are well-suited to solve graph analytics problems, as these problems are characterised by frequent and fine-grain synchronisation [4, 54, 72, 80, 85, 108]. Hence, this dissertation investigates graph analytics in shared memory systems.

Non-Uniform Memory Access (NUMA) Architectures. The terabyte-sized main memory implies that NUMA architectures should be considered for shared graph analytics models. NUMA provides separate memory for each CPU. Each CPU can access its own local memory faster than non-local memory, which is local to another CPU or shared between CPUs. A NUMA-aware graph partitioning can speedup graph analytics, as it reduces remote and random accesses and balances cross-node interconnect bandwidth by optimising access patterns across NUMA nodes [108]. NUMA-aware data placement and code schedule are essential to speedup graph analytics in large shared memory systems.

Hence, the goal of this dissertation is to optimise load balance and memory locality of graph analytics in NUMA-aware shared memory systems. In order to show the benefits of our optimisations, this dissertation selects two comparable state-of-the-art shared memory graph ananlytics frameworks, **Ligra [85] and Polymer [108]**. Ligra is a simple and lightweight graph analytics framework. It shows out-performance and a simple parallelism method compared to the other state-of-the-art shared memory systems. Polymer extends Ligra using graph partitioning and NUMA-aware data access design. It outperforms art-of-the-state shared memory graph analytics frameworks, e.g., GraphChi [54], X-stream [80] and Galois [72]. These two frameworks are important prior work in graph ananlytics. Hence, this dissertation selects them as comparisons. Section 1.1 defines all problems and challenges of graph analytics in this dissertation. Section 1.2 summaries all contributions of this dissertation, and gives an overview of the following chapters.

1.1 Problem Definition

Graph partitioning is an important component to identify independent tasks to maximise parallelism across distributed nodes [40, 41]. It allows to stage graph data in main memory from backing disk [38, 54]. Graph partitioning also allows to distribute graph data in compute clusters [33] and direct memory accesses to the locally-attached memory node in NUMA machines [108]. It can balance the number of vertices or edges per partition. In order to achieve efficient graph analytics, graph partitioning is a key technique to solve the previous mentioned challenges in this dissertation.

1.1.1 Load Imbalance of Graph Partitioning

In an edge-balanced graph partitioning, the edge set is partitioned to ensure that each partition can have an almost equal number of edges. This aims to balance the computation per partition as many graph analyses perform work proportional to the number of edges [33]. However, partitioning the edge set results in an imbalance in the number of vertices appearing in each partition. Alternatively, partitioning the vertex set results in an imbalance in the number of edges across partitions. Thus, significant load imbalance exists between partitions, either for loops iterating over vertices, or for loops iterating over edges [90]. It is important to know about the various partitioning ways and distinction of graph algorithms in which it affects load imbalance performance.

1.1.2 Memory Locality

Three types of memory locality are commonly discussed in graph analytics. The first one is **spatial locality**, which refers to the use of data elements within relatively close storage locations [99]. Graph analytics has a poor spatial locality, as it performs many sparse accesses to small objects [70].

The second one is **temporal locality**, which refers to the reuse of specific data within a relatively small time duration [99]. Graph analytics has a poor temporal locality, as it performs random accesses due to irregular structure of graphs. It is difficult to predict and optimise *reuse distance*, which is the number of distinctive data elements accessed between two consecutive uses of the same element [22, 23, 111].

Finally, low-diameter and power-law distribution of graphs result in the difficulty of effective partitioning [9, 33]. It is complex to find a minimise cut across NUMA nodes. Hence, it results in an inter-socket communication, which has random and remote accesses across NUMA nodes. It is a challenge to reduce communication across NUMA nodes to improve **NUMA locality**.

1.1.3 Graph Layout

There are three common graph layouts of graph analytics: Compressed Sparse Columns (CSC) [13, 81], Compressed Sparse Rows (CSR) [81, 89] and Coordinate list (COO) [81, 82]. Representations such as CSC and CSR effectively provide an index into the edge list, allowing efficient lookup of the edges incident to active vertices [85]. COO lists all edges as a pair of source and destination vertices [81].

CSC sorts all destination vertices in an index order array, which associates to

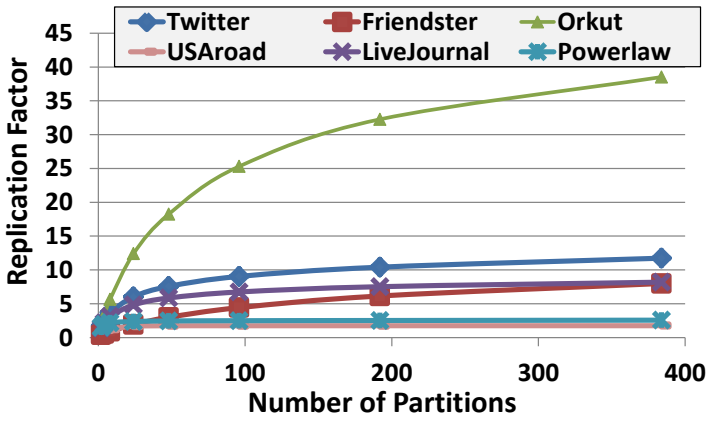


Figure 1.1: The replication factor for a number of graphs and varying number of graph partitions. Results hold for partitioning by destination [90, 108].

the source vertices by incoming edges. CSC traverses all destination vertices, obtains values from active source vertices sequentially by following incoming edges. CSR sorts all source vertices in an index order array, which associates to the destination vertices by outgoing edges. CSR traverses all active source vertices, updates values to the destination vertices by following outgoing edges. COO requires traversing all edges. For CSR and COO layouts, multiple edges with same destination vertices of a graph result in an update of same vertex may happen at the same time. It is essential to use *hardware atomic operations* [77], which load-add-store values in a single step, to avoid data race during parallelism. The choice of graph layouts requires the programmer to decide whether an algorithm runs fastest when traversing the graph in CSC, CSR or COO.

Implementing a large number of partitions requires careful design of the graph data structures. When partitioning the edge set, in CSR and CSC layouts, some vertices will appear in multiple partitions. These are called *replicated vertices* [33] or *ghost vertices* [108]. The replication factor quantifies the number of partitions where a vertex is replicated because it has incoming edges or outgoing edges that

are assigned to that partition. A replication factor larger than one is a logical consequence of edge partitioning [33]. Figure 1.1 shows that the replication factor already becomes significant for 384 partitions: 11.7 for Twitter and 38.5 for Orkut. Hence, CSR or CSC does not scale to a large number of partitions due to either edges or vertices crossing partitions, requiring replication of those edges or vertices in all relevant partitions [33]. Graph storage is limited by the increasing number of partitions. COO does not replicate vertices. However, COO requires all edges must be traversed during each iteration of graph algorithms, which is prohibitively expensive when few edges are active [4, 43, 108].

1.1.4 Graph Ordering

Prior researches speedup graph analytics using graph ordering [11, 31, 42, 49, 98, 112]. Graph ordering algorithms do not require specific optimisations on graph algorithms or graph data structures. However, they do not always work well, due to the NP-hardness of solving the exact problem [24]. Intuitively, edge balance and edge cut minimisation are partially contradictory constraints. Moreover, few graph reordering works mentioned above investigate NUMA-aware architecture and load balance together.

1.2 Thesis Contribution and Structure

Chapter 2 presents the most strongly related work. The other chapters present as the following contributions:

- **Chapter 3.** Chapter 3 shows that heuristic edge-balanced partitioning results in an imbalance in the number of vertices per partition. Thus, significant load

imbalance exists between partitions. To address this issue, Chapter 3 introduces a classification of algorithms to distinguish whether they algorithmically benefit from edge-balanced or vertex-balanced partitioning. This classification supports the adaptation of partitions to the characteristics of graph algorithms. This classification is implemented in GraphGrind, a NUMA-aware graph analytics framework that improves NUMA locality by matching the number of partitions to the number of NUMA nodes, and mapping key data structures across NUMA nodes. GraphGrind is compatible with the Ligra [85] programming model to achieve excellent spatial locality by sequential accesses to the edge list of a vertex. This chapter is published in the ACM/SIGARCH International Conference on Supercomputing (ICS-2017).

- **Chapter 4.** This chapter presents that GraphGrind improves temporal locality [89] by assigning all incoming edges of a vertex to the same partition [54, 108]. It demonstrates that *reuse distances* are reduced as the graph is more finely partitioned. Furthermore, this chapter investigates how far GraphGrind can scale graph partitioning in order to maximally benefit from temporal locality. It confines all updates to a value to one partition and one thread, which boosts performance by avoiding hardware atomic operations. Load balancing work across threads as the amount of work per partition varies with graph structure and active edge set. It resorts to using a graph layout that is immune to vertex replication and designs a fully automated graph traversal algorithm that extends the previously established graph traversal directions heuristic to a 3-way graph layout choice. This new algorithm furthermore depends upon the classification of graph algorithms presented in Chapter 3. This chapter is

published in International Conference on Parallel Processing (ICPP-2017) by IEEE.

- **Chapter 5** Chapter 3 and Chapter 4 use the adaptation of partitions to the characteristics of graph algorithms to deal with load imbalance. Chapter 5 identifies that the time for processing a graph partition depends on both the number of edges and the number of unique destinations in that partition. Hence, it introduces an efficient graph ordering algorithm to partition graphs through joint destination vertex- and edge-balanced partitioning to deal with load imbalance. It does not need to classify the partitioning method based on the characteristics of graph algorithms identified in Chapter 3.

Above all, graph partitioning can optimise *load balance*, *NUMA locality and temporal locality* of graph analytics in shared memory systems. Finally, Chapter 6 makes a general conclusion and discusses open issues.

Chapter 2

Background

This chapter introduces the most notable state-of-the-art frameworks and optimisation techniques of graph processing that are used in this dissertation.

2.1 Graph Processing

Graph algorithms perform iterative computations on values associated to the edges or vertices of a graph, until a convergence criteria is met, or a fixed number of iterations have been performed. Hence, graph processing contains a graph representation and graph algorithms perform graph traversal. These affect the performance of graph processing.

2.1.1 Graph Representation

In each iteration, graph computation needs the following data structure.

- A *graph* $G = (V, E)$ has a set of vertices V and a set of directed edges $E \subset V \times V$ represented as pairs of end-points.

- A *frontier* is a subset of the vertices which are active to propagate values, $F \in \mathcal{P}(V)$ (power set).
- *Active edges* are outgoing edges of the vertices of the *frontier*, $(u, v) \in E : u \in F$.

In each computing iteration, graph processing traverses a current frontier, and visits the destination vertices of the active edges to apply an algorithm-specific function to update the value computed for v taking into account the current value for u. At the end of each iteration, graph processing calculates a new frontier that consists of the updated vertices status, which becomes the frontier in the next iteration. In general, graph algorithms iterate the same procedure until convergence, which is typically signified by an empty frontier.

The frontier may be *sparse* or *dense* [43]. A sparse frontier contains few active vertices. A frontier is typically considered sparse if the sum of active vertices and active outgoing edges is fewer than a threshold, such as 5% of the edges of a graph [85]. In a *dense frontier* more *edges* are active, Ligra shows that a sparse frontier does not need to traverse all the vertices because of a few active vertices [85]. A sparse frontier is typically represented as a list of vertex IDs [85]. A dense frontier is represented as a bitmap. The sparse frontier will be traversed by iterating only over the active vertices, while the dense frontier will be traversed by iterating over all vertices and checking for each of them whether they are active.

2.1.2 Graph Layouts

Graph processing needs a set of values associated with edges or vertices of graphs during computation. It contains graph topology data, application-defined data

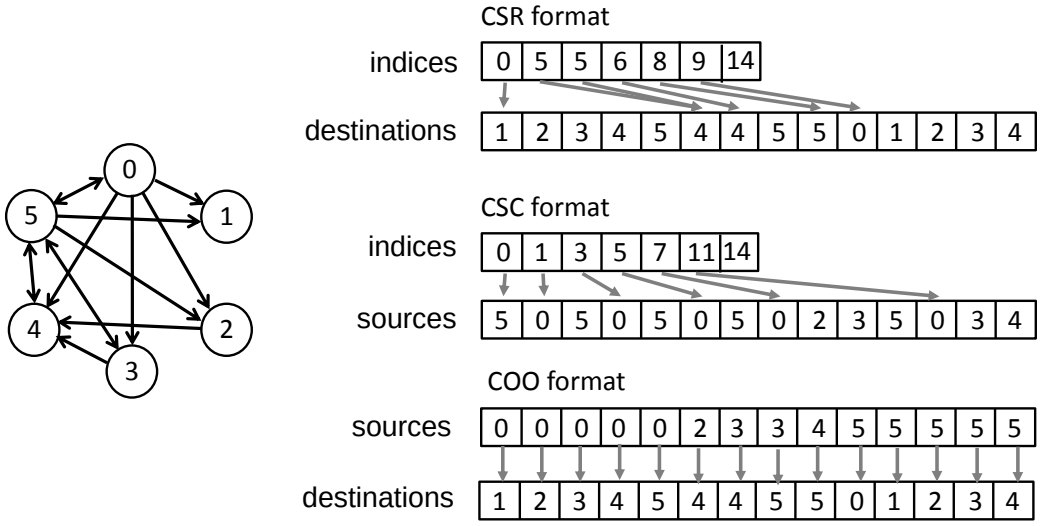


Figure 2.1: Compressed Sparse Rows (CSR), Compressed Sparse Columns (CSC) and Coordinate list (COO) graph data structure

and graph runtime states. These data need to be stored in an appropriate data structure. Figure 2.1 illustrates a graph with skewed degree distribution and three state-of-the-art graph data structures, CSR [81,82], CSC [13,81] and COO [81,89]. In Figure 2.1, these three graph data structures represent the same graph as on the left, and they have their corresponding graph traversal method during computation.

- *COO* format lists all edges of a graph as a pair of source and destination vertices. During traversing, COO traverses all edges and checks whether the source vertices are active, if active, it updates computation values and status to the destination vertices following edges. There is data race due to multiple source vertices update to same destination vertices simultaneously.

- *CSR* format contains two arrays: (i) an edge array with IDs of the destination vertices, (ii) an index array storing for each vertex the index into the edge array where the destinations of its edges are recorded. It traverses all active

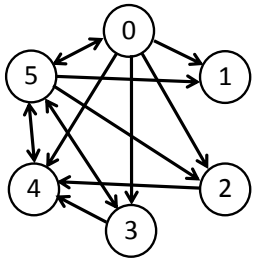
source vertices and update the values and status to their destination vertices following outgoing edges. There is data race due to multiple edges of vertices, several source vertices may be updated to same destination vertices at the same time.

- *CSC* format contains two arrays: (i) an edge array with IDs of the source vertices, (ii) an index array storing for each vertex, the index into the edge array where the sources of its edges are recorded. It traverses all destination vertices, and then checks whether their source vertices are active following the incoming edges sequentially. If active, it updates the computation values and status to the destination vertices following incoming edges.

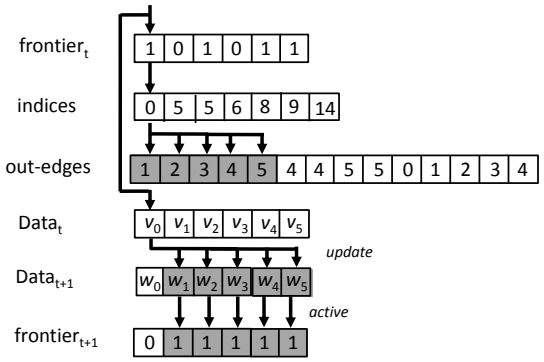
The index array of CSC and CSR has length $|V| + 1$, and the edge array has length $|E|$. The edge array of COO has length $2|E|$. Graphs may be traversed in either *CSR*, *CSC* or *COO*. It is difficult to distinguish which graph layout is fastest. Beamer *et al.* [8] present that a number of algorithms execute faster when using CSC order. Since the data race in CSR and COO needs *atomic* operations [85] to ensure correctness, it results in a high hardware overhead and a slower performance compared to CSC format in some algorithms [85, 108]. In general, the programmer needs to determine experimentally when an algorithm executes fastest with CSR, CSC or COO [85].

2.1.3 Graph Traversal Models

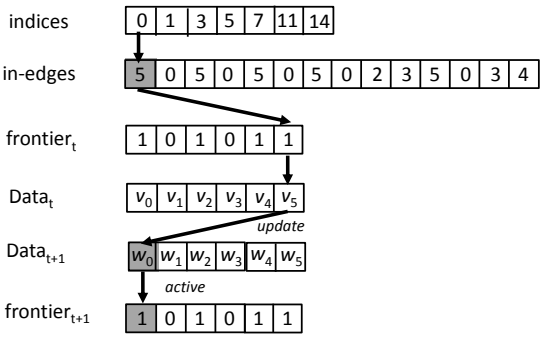
The **vertex-centric** model traverses list of active vertices sequentially but has random access on edges, which is suitable for CSC and CSR layouts. It has two traversal approaches, *forward (push)* or *backward (pull)* [33, 59, 77].



(a) graph



(b) Forward using CSR



(c) Backward using CSC

Figure 2.2: Graph traversal operators in CSR and CSC layouts

- Forward (push) traverses active vertex array, pushes updated value to all target vertices by following outgoing edges, and enables those vertices for the next round. It is similar to CSR layout traversal.

- Backward (pull) traverse all vertices, obtains values from active vertices following incoming edges, updates value and state (active) of vertex for next round. It is similar to CSC layout traversal.

Figure 2.2 presents *forward and backward* traversal operators of Figure 2.2a. Figure 2.2b shows that *forward* traverses source vertices $u \in V$ and checks if they are active ($u \in F$) in frontier at first. Vertex 0 is active in $frontier_t$, its destination vertices 1-5 will be updated the values in $Data_{t+1}$ and statuses in $frontier_{t+1}$

that are used for next iteration. Figure 2.2c shows that *backward* iterates over all destination vertices $v \in V$ as well as their incoming edges $(u, v) \in E$. Vertex 5 is the active destination of vertex 0, the update value and status of vertex 0 will be written to $Data_{t+1}$ and $frontier_{t+1}$. Based on these two traversal methods, some state-of-the-art graph analytics frameworks [85,108] propose an abstraction used to switch the traversal methods for faster execution time.

The **edge-centric** model traverses list of edges sequentially but accesses vertices randomly [73, 80]. This is like graph traversal using COO layout. In the scatter phase, it iterates over edges and updates from the current data array. It reads values from source vertices and updates values to destination vertices. Multiple edges with same destination vertices of a graph result in an update of same vertex may happen at the same time. It is necessary to use appropriate concurrent constructs such as *lock*s or *atomic options* [77, 80] to avoid data race. X-stream [80] reduces random accesses to vertices by several approaches, such as shuffle phase.

2.1.4 SYNC or ASYNC

The **SYNC (synchronous)** model traverses all active vertices, then applies updates to the values and move on to next iteration. It is Bulk Synchronous Parallel (BSP) programming model [93], such as Pregel [63]. Each vertex of Pregel runs a graph algorithm and sends messages asynchronously to the other vertices. Pregel uses supersteps to divide computation with a global synchronisation barrier [52]. Each vertex of each superstep processes incoming messages from the previous superstep until all vertices have no messages to send [52,63]. The **SYNC**

model prefers to process larger graphs under limited computing resources [100]. It prefers a large number of updates per iteration, hence, the overhead of a global barrier is minimised [73, 100]. The **ASYNC (asynchronous)** model traverses any active vertex, even if it has been visited recently, and updates values directly. Hence, the same vertex is visited many times. The computation of a vertex uses the new state of adjacent vertices without a global barrier. The **ASYNC** model is used for timely visibility to update [100]. It presents a fast convergent performance by sufficient computing resources. However, message communication happens at any time between different machines results in the delay of updates. The random read/write for same vertex data needs *atomic* operations to avoid data race, which results in scheduling overhead [73, 100]. This overhead depends on the number and degree of active vertices. The increase of number or degree of active vertices results in a large lock contention overhead. Hence, asynchronous is suitable for graphs with fewer degree of active vertices.

Most algorithms are synchronous, few are naturally asynchronous. Xie el at. [100] proposes *Hsync (hybrid)*: switches dynamically between *synchronous* and *asynchronous* based on the rate of vertices and edges processed. The mode switches based on the characteristics of graph algorithms. Some algorithms have a large number of active vertices, such as PageRank [74], therefore it prefers *synchronous* with less inter-node communication [73]. While others have a few active vertices, such as Single Source Shortest Path (SSSP) [17, 103]. It propagates high changes in a vertex's shortest path in SSSP very quickly, hence the *asynchronous* is a better choice.

Algorithm 1 Ligra-edge-map

input : Graph $G = (V, E)$; frontier F; function Fn; bool Fwd
output : New frontier containing updated vertices
side effect: Fn applied to $(u, v) \in E : u \in F$

```
1 if #{(u,v) ∈ E : u ∈ F} > threshold then                    // Dense frontier
2 |   if Fwd then return edgeMapDenseForward(G_p, F, Fn);      // CSR format
3 |   else return edgeMapDenseBackward(G_p, F, Fn);           // CSC format
4 else
  |                                                          // Sparse frontier
  |   return edgeMapSparse(G, F, Fn)                          // CSR format
5 end
```

2.1.5 Programming Abstraction

Graph analytics can use an abstract to unify models and features for multiple graph algorithms. It may be an *vertex-centric* or *edge-centric* model for iterative computation [72, 85]. It may be traversed in *forward* or *backward* manner. The efficient implementation of graph algorithms is sophisticated and requires deep knowledge of the characteristics of the algorithms and graph layouts. For example, the frontier may be implemented either as a bitmap or as an array storing vertex IDs. Beamer *et al.* [8] show that most efficient frontier implementation depends on the *density* of the frontier [43].

Algorithm 1 shows that Ligra, an asynchronous model, uses an *edge-map* function to do a traversal operation switch depends on density of frontier [85]. The *Edge-map* updates the vertex status and value in response to an edge using forward or backward. However, it needs to supply two versions of application-specific update function for thread-safe (backward) or non-thread-safe (forward) [73]. Ligra uses corresponding update function based on whether it executes forward or backward traversal. The *Vertex-map* updates vertex values and status without relevant to edge.

The selection of *backward and forward* depends upon which operator is faster. The distinction is to a large extent motivated experimentally [85]. Beamer *et al.* motivate the distinction by the number of visited edges [8]. The graph representation is designed for efficient forward *and* backward iteration. Hereto, a dual representation is used for directed graphs (incoming and outgoing edges are equal for undirected graphs), i.e., the graph is stored once in CSC format and once in CSR format [85].

2.1.6 Graph Algorithms

Optimising specific graph algorithms for specific frameworks [39, 45, 78] is a common way to speedup graph analytics. However, it requires more efforts to repeat optimisation techniques, for instance, the way to distribute data. A general purpose framework is essential so that the programmer does not need to repeatedly deal with some challenges. Hence, this dissertation uses the basic algorithm APIs from Ligra [85] and Polymer [108] without using specific optimisations. The characteristics of the following graph algorithms are in the Table 2.1.

- Betweenness Centrality (BC) [26] is a useful indicator of the relative importance of nodes in a graph [85]. BC is a measure of centrality in a graph based on the shortest paths. For each pair of vertices in a connected graph, there exists at least one shortest path between the vertices. Hence, there is a minimised number of edges that the path will pass through.

- Connected Components (CC) [17] tries to find connected components subgraphs of a graph. There is no path between vertices which are belonging to different components. One connected component of an undirected graph is a

subgraph. In this subgraph, any two vertices are connected to each other by a path, and which is connected to no additional vertices in the supergraph [79].

- PageRank (PR) [74] computes the rank value of each vertex based on the rank values of its neighbours. It computes the relative importance of webpages [12]. In general, PageRank computations require several iterations, through the collection to calculate an approximate PageRank values, which will closely reflect the theoretical true value.

- Breadth-First Search (BFS) [8] is an algorithm for traversing or searching a tree in a graph. It starts at a root vertex and explores its neighbour vertices first, before moving to the next level neighbours

- PageRankDelta (PRDelta) [59, 85] is a variant of PageRank implemented in Ligra [85]. The vertices are active in an iteration only if they have accumulated enough change in their PageRank value.

- Sparse Matrix-Vector Multiplication (SPMV) [47] multiplies the sparse adjacency matrix of a directed graph with a dense vector of values (weighted values) [108].

- Bayesian Belief Propagation (BP) [48,75] estimates the probabilities of vertices by message passing iteratively between vertices along edges with weighted values. It is widely applied in artificial intelligence and information theory.

- Bellman-Ford (BF) [17] finds a single-source shortest path of a weighted graph from a specific vertex. During graph traversal, it computes the shortest path distance from start vertex to each vertex of the graph.

Table 2.1: Graph algorithms and their characteristics. Traversal direction is used by Ligra and Polymer. Vertex-/Edge-orientation is used by GraphGrind. Frontiers are dense (d), medium-dense (m) or sparse (s).

Code	Description	Traversal direction	Vertex/Edge orientation	Frontier density
BC	betweenness-centrality [85]	backward	vertex	m/s
CC	connected components using label propagation [85]	backward	edge	d/m/s
PR	simple Page-Rank algorithm using power method (10 iterations) [74]	backward	edge	d
BFS	breadth-first search [85]	backward	vertex	m/s
PRDelta	optimised Page-Rank forwarding delta-updates between vertices [85]	forward	edge	d/m/s
SPMV	sparse matrix-vector multiplication (1 iteration)	forward	edge	d
BF	Bellman-Ford algorithm for single-source shortest path [85]	forward	vertex	d/m/s
BP	Bayesian belief propagation [108] (10 iterations)	forward	edge	d

2.2 Graph Datasets

This dissertation selects eight graphs as graph datasets. Twitter is a real-world social network graph containing 41.7 million vertices and 1.47 billion directed edges [53]. Friendster is a real-world on-line gaming network graph from Stanford Network Analysis Project (SNAP) [104]. Orkut [69] is a social network graph where users form friendship each other. LiveJournal [104] is a real-world on-line community to allow members to maintain journals and allow people to declare which other members are their friends they belong. Yahoo_mem is a real-world graph of the Web members from [94]. USAroad is a road network of the United States from the 9th DIMACS shortest paths challenge [108] with much

Table 2.2: Characterisation of real-world and synthetic graphs used in experiments.

Graph	Vertices	Edges	Max. Degree	Type
Twitter [53]	41.7M	1.467B	770,155	directed
Friendster [104]	125M	1.81B	4,223	directed
Orkut [69]	3.07M	234M	33,313	undirected
LiveJournal [104]	4.85M	69.0M	13,906	directed
Yahoo_mem [94]	1.64M	30.4M	5,429	undirected
USAroad [108]	23.9M	58M	9	undirected
Powerlaw ($\alpha = 2.0$)	100M	1.5B	2758	directed
RMAT27	134M	1.342B	812,983	directed

high diameter. Powerlaw and RMAT27 graphs are generated by Problem Based Benchmark Suite.(`http://www.cs.cmu.edu/~pbbs/`). These graphs are popular in the art-of-the-state work.

2.3 Graph Analytics Frameworks

Several state-of-the-art frameworks [1,33,34,37,52,59,63,83,100,107] have shown distributed memory systems have a powerful ability to deal with the increasing scale graph data with adding more processing nodes to the system [68]. On distributed systems, graph data can be partitioned over the separate nodes. The inter-communication between nodes are used for synchronising computation. Communication cost between machines is increasing quickly. High communication overhead results in a bottleneck of fast execution time. Shared memory systems have a limited scalability, but multiple tasks can access the same memory with lower communication cost. They do not require programmers to manage cluster communication or task parallelism with a relative flexible way. Table 2.3 introduces some state-of-the-art designs in the distributed, disk-based and shared

memory systems, which are discussed in this section. These frameworks are popular and relevant to load balance and memory locality of graph analytics.

2.3.1 Distributed Memory Systems

Distributed memory systems (scale-out) are mostly used for large scale graph analytics. Graph data can be partitioned over distributed nodes for parallelism.

Pregel [63] is a vertex-centric distributed graph processing framework, it is efficient, scalable and fault-tolerant on clusters. It is a synchronous model, which ensures the lower cost for imbalanced work to avoid faster workers having to wait frequently. Pregel mainly focuses on sparse graphs, the communication cost usually happens over edges. Programmers implement their algorithms into Pregel using *combiner, aggregator, and topology mutations*. Pregel is still a message passing model, which requires highly communication control, if graph is dense, the overhead will be high.

Giraph [1] is inspired by Pregel but running on Hadoop, it uses "thinking like a vertex" [65] method to process vertex in parallel. Compared to Pregel, Giraph reduces memory usage and computation time. Each worker works with multiple graph partitions, and maintains its own message to hold all incoming messages. In order to reduce contention on the storage and efficiently utilise network resources, each compute thread has a message buffer cache to deal with all outgoing messages [36]. Hence, Giraph has a high overhead on communication.

Distributed GraphLab [59] extends shared memory GraphLab using graph partitioning to reduce network congestion. Each partition is executed on a separate machine on a cluster. It uses *edge-cut* to partition graphs where the partition

Frameworks	Distributed (D) or Shared (S)	NUMA awareness	Forward (F) or Backward (B)	Graph Partitioning (Stream Partitioning(SP))	Load Balancing
Pregel [63]	D	No	F	Edge-cut and SP	No
Giraph [1]	D	No	F	Edge-cut and SP	No
PowerGraph [33]	D	No	F/B	Vertex-cut and SP	No
PowerSwitch [100]	D	No	B	Vertex-cut and SP	No
GPS [83]	D	No	F	Edge-cut and user selects SP	Implicit Support
Gaffer [2]	D	No	F	SP	Implicit Support
GraphX [34]	D	No	F	Vertex-cut and SP	Implicit Support
Distributed GraphLab [59]	D	No	F/B	Edge-cut and SP	No
Chronos [37]	D	No	F/B	Edge-cut and SP	No
Mizan [52]	D	No	F	Edge-cut and SP	Yes
GraphChi [54]	Disk-based	No	B	Edge-cut	No
TurboGraph [38]	Disk-based	No	F	Edge-cut	No
Graspan [38]	Disk-based	No	F	Edge-cut	No
GraphLab [60]	S	No	B	No	No
X-stream [80]	S	No	F	Vertex-cut and SP	No
Galois [72]	S	Yes	F/B	Edge-cut and SP	Yes
Ligra [85]	S	No	F/B	No	No
Polymer [108]	S	Yes	F/B	NUMA-Aware vertex-cut partitioning	No
Grace [102]	S	No	F	Edge-cut	Yes
Ringo [76]	S	No	F/B	No	No

Table 2.3: State-of-the-art graph analytics frameworks in the Section 2.3.

boundary is a set of edges. To ensure safe parallelism, it enforces a consistency and hides network latency using the pipelined locking. If a lock of an edge or vertex happens on the boundary, it locks on both partitions, which is distributed locking. Distributed GraphLab benefits from the asynchronous and pipelined locking, but still has an issue of communication overhead.

PowerGraph [33] is a vertex-centric framework on natural graphs using the *Gather-Apply-Scatter (GAS)* model. Natural graphs are usually sparsity connected, in which most vertices have few neighbours while a few have many neighbours, a power-law degree distribution, such as twitter follower graph [53]. It is difficult to partition the power-law graphs because they do not have low cost balanced cuts in a general way. PowerGraph splits high degree vertices and parallels high-degree vertices for an equivalence on split vertices. It uses vertex-cut to ensure that each partition contains a subset of edges in equal. They compute over edges instead of vertices with supporting both synchronous and asynchronous scheduling.

PowerSwitch [100] supports adaptive switches between *sync and async* modes for optimal performance. It proposes *Hsync* built on PowerGraph. It allows an efficient sharing of states between two modes for a safe and automatic mode switch at an appropriate time. *Sync* mode reduces the message communication overhead and improves the hardware utilisation. *Async* model accelerates convergence speed under sufficient hardware support. However, performance is limited by the applications, for instance, if an algorithm is relative stable, which is without workload change or frequent communication request, there is no need to switch.

Graph Processing System (GPS) [83] is a scalable, fault-tolerant vertex-

centric synchronous distributed graph analytics framework. It uses a dynamic repartitioning approach to assign vertices to different workers during graph computation based on message passing patterns. It also provides an optimisation that distributes adjacency lists of high-degree vertices across all compute nodes. It tries to deal with the load imbalance issues across separate nodes.

Gaffer [2] is a large scale database of relation and entity supporting aggregation of properties, build in Spark [107]. It is synchronous model. It aims at providing a flexible, scalable and extensible computation for large scale graphs. Hence, it allows for rapid prototyping and transition to production systems with querying across large amount of entities and relationships quickly. In additions, it allows filtering and transformation of data rapidly, the scalability aims at very high data rates and volumes, automated.

GraphX [34] is a graph-centric synchronous distributed framework. It adds graph indexing and partitioning on the top of Spark's resilient distributed datasets (RDD) [107]. It is an integrated graph representation with collections API. GraphX uses data flow optimisations in graph processing systems. It uses vertex cut partitioning to reduce communication cost because a vertex may have many edges in the same partition.

Chronos [37] is a vertex-centric synchronous graph engine for temporal graph analysis in distributed systems. It optimises in-memory structure iterative graph computation on temporal graphs. Locality is the key contribution of Chronos, they pay attentions on the design of in-memory layout and the scheduling of iterative computation on temporal graphs, so that traversal operations are scheduled to maximise the benefit of in-memory data locality.

Mizan [52] shows that graph partitioning is insufficient for minimising end-

to-end computation, which has been built on Pregel, specially when data is very large or the runtime behaviour of the algorithm is unknown. It needs an adaptive method. Mizan achieves efficient load balancing to adapt changes in computing needs with monitoring the runtime characteristics of the system. Mizan also performs efficient fine-grained vertex migration to balance computation and communication.

These distributed graph analytics frameworks tries to reduce the message passing overhead. They run efficiently on small-world graphs. Most distributed frameworks discussed in this dissertation are synchronous models. Xie *et al* [100] present that synchronous model is not suitable for graph algorithms which computation converges asymmetrically, such as BFS. It is also not suitable for algorithms that require coordination between adjacent vertices [73, 100], such as Graph Colouring [32], all vertices need synchronous change to the same colour [73, 100]. These distributed frameworks do not work well for all algorithms. Compared to shared memory systems, distributed systems are less dependent on CPU evolution for scaling, but communication cost between machines becomes a bottleneck in large scale graph analytics.

2.3.2 Disk-Based Systems

GraphChi [54] is the first single machine framework to use external memory using an optimised format to store graph on disk. It presents an efficient disk-based vertex-centric asynchronous framework. It proposes a parallel sliding windows (PSW) for large graphs on a single machine. There are three main stages after graph partitioning: 1) load subgraph from disk, 2) update the values of vertices

and edges, and then 3) write the updated values to disk. In the second stage, some critical vertices which have edges with both end-points in the same interval, they need to be executed in sequential order to avoid data race. GraphChi has limited parallelism and separate steps for I/O processing and CPU processing.

TurboGraph [38] is a synchronous matrix computation model with edge-cut partitioning. It exploits full parallelism including multi-core parallelism and FlashSSD IO parallelism compared to GraphChi. It proposes pin-and-slide model, which is a new computing approach for efficient processing the generalised matrix-vector multiplication in the column view.

Graspan [96] is an edge-centric repartitioning computation model for inter-procedural static analyses. Small graphs can be held in memory, and partitioned into two partitions. Large graphs will be partitioned into more than two partitions. Graspan checks edge duplicates quickly before an edge is added. However, it has additional execution time for graph checking and repartitioning.

Disk-based systems use external memory to deal with graphs that are too large to fit in main memory [73]. They optimise graph format on disk, so the frameworks can work with graphs that even do not fit into main memory. However, they are still inefficient due to redundant I/O accesses and poor scalability of parallelism.

2.3.3 Shared Memory Systems

GraphLab [60] is a vertex-centric asynchronous shared memory graph analytics framework. It offers scalable performance using combination of specific computation, data-dependencies, and scheduling. GraphLab provides a set of data

consistency models, such as full consistency, edge consistency and vertex consistency, which help users to specify the minimal consistency requirements of their application without building their own complex locking protocols.

X-stream [80] uses edge-centric for a scatter-gather model. It is synchronous model. It borrows the idea from distributed systems to do the edge partitioning. It uses streaming edge partitioning to avoid random access to edges, and gets benefits if the edge set is much larger than the vertex set. It improves the spatial locality to ensure the sequential access. After partitioning, the vertices have been partitioned to fit in memory.

Galois [72] is data centric framework, which uses both synchronous and asynchronous models. It focuses on abstraction around algorithms. It determines that algorithm properties are important for parallelism by expressing behaviours on data structures. Galois has several dynamic elements to build graphs dependence and scheduling, such as data driven or topology driven. It proposes an optimistic parallelism and asynchronous mode that to consider an algorithm coverage for ordered, data driven algorithms.

Ligra [85, 87] is a simple and lightweight graph analytics framework using vertex-centric. It is synchronous model, but it allows to use prior updates from the same superstep. It provides two graph traversal functions, edgeMap and vertexMap, which will map the computation over edges or vertices of the *frontier* iteratively. Ligra provides a switch for selecting different traversal method, *backward or forward*. It also proposes a dense and sparse frontier with different data structures to represent. Ligra uses the whole graph to do the computation, so it treats the vertices as a parallel unit.

Polymer [108] is a NUMA-aware vertex-centric synchronous graph partition-

ing framework. It tries to minimise both random and remote memory accesses by optimising graph data layout and access strategies for subgraphs across NUMA domains. Polymer does the graph partitioning over NUMA domains, where each partition has almost same amount of edges. Each NUMA node could do subgraph graph computation individually in parallel. This reduces the remote access across NUMA nodes and the traversal costs.

Grace [102] is asynchronous vertex-centric stream partitioning model. It attempts to improve in-memory bandwidth utilisation. It uses a scheduler to control the order of vertex computation in a block. It contains a number of graph-specific and multi-core-specific optimisations including graph partitioning, vertex ordering and load balance. Grace investigates a simple graph querying interface to build a framework to run iterative graph computations.

Ringo [76] is an interactive asynchronous graph analytics framework. It is interactive with allowing raw input data to be manipulated into graphs and has over 200 graph analytics applications. It represents graph data in a hash table of nodes to support dynamic graphs.

Shared memory systems have an efficient intra-communication that multiple processing units can access and update the same memory. Even though scalability of shared memory (scale-up) is limited, communication cost is lower than distributed systems. Synchronous model is not suitable for asymmetric convergent algorithms. The Asynchronous model is not suitable for graphs with high degree of vertices. A framework with synchronous and asynchronous models together, such as PowerSwich, needs to check the characteristics of graph algorithms first. It results in high execution time.

Shared memory systems do not require the programmer to be skilled at man-

aging clusters and task parallelism, and the programming environment is relative flexible. Even though they have a memory limitation, shared memory systems with terabyte-sized main memory provide an opportunity to do efficient graph analytics. Hence, this dissertation is mainly investigating graph analytics on large shared memory systems. Chapter 1 presents two main challenges in graph processing, load imbalance [61] and poor locality [61, 73]. Hereto, this dissertation aims to investigate how shared memory systems are well-suited for graph analytics with dealing with load imbalance and poor locality.

Moreover, Ligra proposes that a synchronous model uses prior updates from same superstep, like the asynchronous model, but saves the time to determine the model direction. It shows a simple parallelism method with out-performance compared to the other state-of-the-art shared memory systems. However, it does not use graph partitioning and NUMA-aware optimisations. Polymer extends Ligra using graph partitioning and NUMA-aware design. It outperforms mostly art-of-the-state shared memory graph analytics frameworks. Hence, this dissertation selects them as comparisons.

2.4 Graph Partitioning

Classifying independent tasks and distribute them for parallelism is essential for high performance. It is difficult to realise independent tasks, as it has to consider the appropriate synchronisation and data consistency. This is the reason why that communication cost must be considered in distributed memory systems. Graph partitioning provides an opportunity to distribute the independent task for parallelism. However, graph partitioning is hard to match a varying degree of

Algorithm 2 Partitioning by destination

input : Graph $G = (V, E)$; number of partitions P
output : Graph partitions $G_i = (V, E_i)$ for $i = 0, \ldots, P-1$

```
6  avg = |E|/P;                                  // target edges per partition
7  i = 0  for v : V do
8      if |E_i| >= avg and i < P − 1 then
9          ++i;                                  // i has exceeded target edges
10     E_i = E_i ∪ in-edges(v);                  // i is home partition of v
```

parallelism of graph computations.

2.4.1 Graph Partitioning Algorithms

Graphs can be partitioned by either partitioning the **vertex set** or the **edge set**. Partitioning the **vertex set** implies that some edges cross partitions, which requires additional reduction or communication steps when processing these edges. Alternatively, one can partition the **edge set**. An often-used criterion for balancing CPU load is to equalise the number of edges per partition, as many graph analytic algorithms perform an amount of work that is proportional to the number of edges. Ghost vertices model replicas of vertices in other partitions [59]. This dissertation focuses exposition on the edge set graph partitioning proposed in GraphChi [54]. GraphGrind partitions the edge set by (i) partitioning the vertex set, (ii) deciding a home partition for each vertex and (iii) partitioning the edge set using the home partition of either the source or destination vertex of the edge. Each vertex is replicated in every partition that holds an edge incident to the vertex. Formally, assume a graph $G(V, E)$ where V is the vertex set and $E \subset V \times V$ is the edge set, the set of vertices V is partitioned in k non-overlapping sets by $\mathcal{P} = P_i, i = 0, \ldots, k-1$. $P_i \subset V$ and $\bigcup_{i=0}^{k-1} P_i = V$ for all i, and $P_i \bigcap P_j = \emptyset$ for all $i \neq j$. This partitioning of vertices supports two

options for partitioning the edge-set:

- **Partitioning by destination**: all in-edges of a vertex are in the home partition of the vertex

$$G_P^{dst} = (V, \{(u, v) \in G.E : v \in P\}) \tag{2.1}$$

- **Partitioning by source**: all out-edges of a vertex are in the home partition of the vertex

$$G_P^{src} = (V, \{(u, v) \in G.E : u \in P\}) \tag{2.2}$$

These partitioning algorithms can be performed with a single pass over the edge list. Algorithm 2 shows the algorithm for partitioning by destination.

2.4.2 Vertex Replication

Chapter 1 introduced that there are *replicated vertices* [33] or *ghost vertices* [108] when partitioning the edge set. Figure 1.1 measures more partitions up to 384 partitions, the replication factor grows slower than a linear function of the number of graph partitions. The worst-case replication factor is $r(|V|) = |E|/|V|$: when every vertex is placed in a distinct partition, then a vertex is replicated once per incident edge. The worst-case replication factor for Twitter is 35.2, while for Orkut it is 76.2. Figure 1.1 shows that the replication factor already becomes significant for 384 partitions: 11.7 for Twitter and 38.5 for Orkut. This effectively makes it appear as if the partitioned Twitter graph has 11.7 times as many vertices as the actual graph, which has adverse impact on memory usage and execution time. It is also relevant to the graph layouts. Vertex replication is a key issue to increase the number of graph partitions.

2.4.3 Graph Storage Size

An immediate consequence of vertex replication is that the graph storage size grows with the number of partitions. This dissertation characterises the storage size for each graph layout assuming directed unweighed graphs to show the impact of graph partitioning. Let $r(p)$ be the replication factor for a p-way partitioned graph. And zero-degree vertices are not stored in the CSR format. GraphGrind stores the vertex ID along with the vertex data in order to save space for zero-degree vertices [90]. The total storage cost for p partitions in CSR format can thus be expressed as:

$$r(p)\ |V|\ (b_e + b_v) + |E|\ b_v$$

where b_e is the storage in bytes for an index in the edge list and b_v is the storage in bytes for a vertex ID. The number of vertices stored grows with a factor $r(p)$.

When both CSC and CSR formats are stored alongside each other [85] to implement the direction-reversing technique, the actual storage size needs to be doubled. Due to partitioning-by-destination does not affect the memory locality of graph traversal, during *backward* CSC traversal, it needs one whole graph CSC format for partitioning computation chunk. The storage size of the CSC format is $|E|b_v+|V|b_e$.

The storage size of the COO format is $2|E|b_v$ and is independent of the number of partitions. The factor 2 results as each edge stores both source and destination vertex ID.

Figure 2.3 shows how storage size varies with the number of partitions. The COO representation is independent of the degree of partitioning and shows a flat line. The CSR representation grows gradually in size, following a curve with the

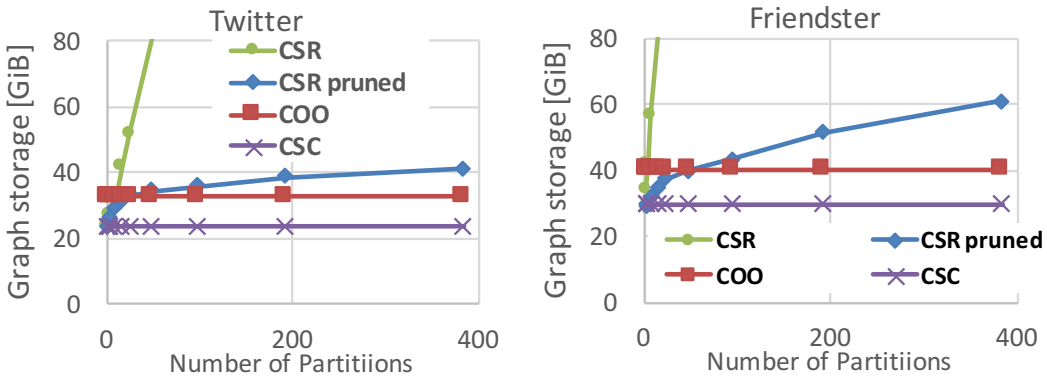


Figure 2.3: Graph storage size for CSC/CSR and COO schemes for the Twitter and Friendster graphs.

same incline as the replication factor (curve "CSR pruned"). Note that while the replication factor of Friendster is smaller than that of Twitter, the Friendster graph has many more vertices. This causes the storage size for the pruned CSR representation to grow more quickly.

When zero-degree vertices are not pruned from the CSR data structure, the storage size grows linearly with the number of partitions as $p\ |V|\ b_e + |E|\ b_v$. In this case, every vertex appears in every partition and the vertex ID need not be stored, reducing the storage per vertex to b_e bytes. Polymer does not prune zero-degree vertices from the representation [108].

This analysis shows that creating many partitions using the CSR formats requires significantly more storage space than working with one partition. This is prohibitive as the available main memory may be considered an inflection point: performance is high if GraphGrind can work within the available main memory; it is poor when it needs to work in an out-of-core manner. As such, one may manage using the CSR layout when sufficient memory is available. However, only the COO layout is scalable to large numbers of partitions.

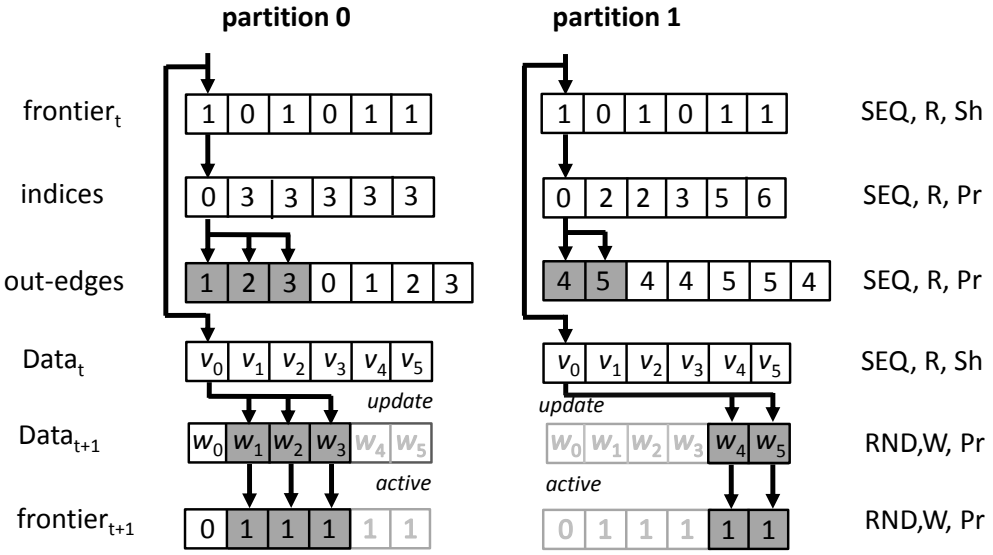


Figure 2.4: Traversal of the graph in Figure 2.2a partitioned into two parts using *partitioning by destination*(Algorithm 2).

2.4.4 Graph Traversal of Graph Partitions

Both partitioning edge-set algorithms achieve nearly the same number of edges in each partition [108]. Figure 2.4 shows how graph partitions do graph traversal individually. Partition 0 contains 7 edges and is home to vertices 0, 1, 2 and 3. Partition 1 also contains 7 edges and is home to vertices 4 and 5. Figure 2.4 furthermore shows how a single traversal of the graph proceeds, assuming a *dense forward* traversal. This traversal first checks Boolean values of source vertices in the frontier whether they are active. For instance, for vertex 0, it traverses the outgoing edges of vertex 0 in each partition in parallel. It computes updated values to vertices 1, 2 and 3 in partition 0 and to vertices 4 and 5 in partition 1. It updates the frontier accordingly. Note that each partition updates distinct values as edges with the same destination appear in the same partition.

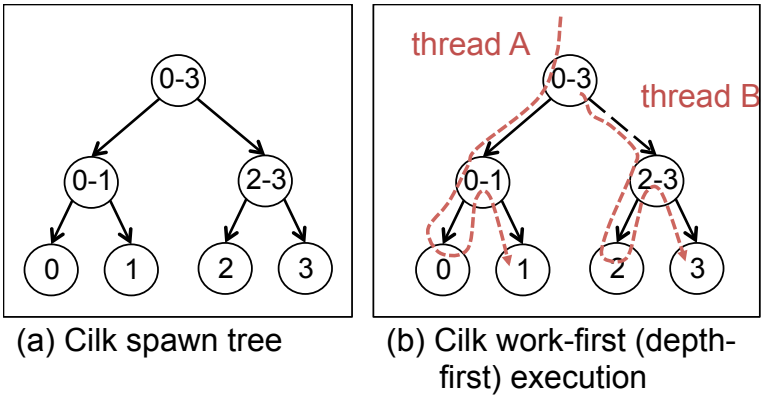


Figure 2.5: Cilk work-stealing.

2.5 Parallelism Tool—Cilk

GraphGrind extends Ligra, which is built on Cilk [29], an efficient work-stealing scheduler for parallel programs. Cilk, however, is agnostic of the memory hierarchy as it promotes cache-obliviousness [28, 105]. Graph analytics processing focuses exclusively on parallel loops, which in Cilk are expressed with the `cilk_for` keyword, asserting that all iterations of the loop may execute in parallel. A minimalistic modification has been deliberately searched for as to not affect space- and time-efficiency [10] and implement this in Intel Cilkplus version 1.2 [46]. A proof of the space and time bounds to future work is delegated.

Cilk implements parallel loops using a helper function that recursively splits the iteration range of the loop in half. Once the iteration range is shorter than a heuristically determined threshold the helper function executes the loop sequentially over this part of iteration range.

Figure 2.5 (a) shows the call tree of the helper function for a loop with 4 iterations. Each node represents an invocation of the helper function. Edges indicate a parent-child relationship between function calls. Nodes in distinct subtrees are

independent and may execute concurrently. Cilk uses a work-first scheduler [10] which translates into a depth-first traversal of the tree (Figure 2.5 (b)). Idle threads attempt to steal work from a randomly selected victim thread. Threads steal the continuation of the oldest function on their victim's call stack, i.e., the one nearest to the root of the call tree. E.g., if thread A starts execution of the range 0-3 in depth-first order it will first execute the sub-range 0-1. Meanwhile, thread B may steal the continuation of the oldest function and execute the sub-range 2-3.

Cilk is a simple and lightweight parallelism language. Programmers do not need to know how to do the efficient parallelism, they can use several keywords to call parallel functions immediately. Hence, this dissertation uses Cilk as parallelism tool and extends Cilk with NUMA-aware optimisation.

2.6 GraphGrind

All the developed techniques of this dissertation are implemented in GraphGrind [89, 90], a NUMA-aware graph analytics framework that builds on the characteristics of graph partitions to optimise the memory layout of graphs and to reduce load imbalance. Figure 2.6 shows that GraphGrind extends Ligra source-code to ensure all the optimisations are behind the interfaces, which ensure programmers to use the techniques easily. GraphGrind contains all the required features of graph analytics systems, including hierarchical parallel decomposition of the computation, NUMA-aware data placement and code scheduling [108], balanced vertex-cut partitioning [33] and adapting data structures [43] and search direction [8] to the size of the frontier. Its key features are discussed below.

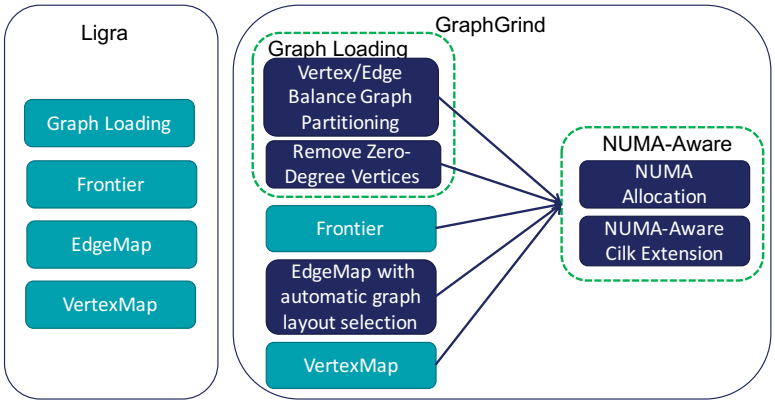


Figure 2.6: Key functions of Ligra and GraphGrind. All the techniques are behind the interfaces. Dark blue blocks are the optimisation of GraphGrind based on Ligra source-code.

2.6.1 Application Programming Interface

GraphGrind is compatible with the Ligra programming model. It provides two data types: graphs and frontiers. The key functions apply operations to edges or vertices and calculate new frontiers in the process. They are defined as follows:

- *size()*: For a frontier F, $size(F)$ returns $|F|$.
- The *edge-map()* operator is the main work-horse. It applies an algorithm-specific function to every active vertex in the graph. Its arguments are a graph $G =(V, E)$, a frontier F, a function Fn and a condition C. An edge $(u, v)\in E$ is active if $u \in F$ and $C(v)= true$. The argument Fwd determines whether a forward or a backward traversal is likely to be faster. *Edge-map* returns a new frontier consisting of all visited vertices v for which $Fn(u, v)$ returned a true value.
- *vertex-map()* applies a function Fn to every vertex in the frontier F. It returns a new frontier consisting of all visited vertices u for which $Fn(u, v)$ returned a

true value.

2.6.2 Parameters of GraphGrind

Figure 2.6 shows that GraphGrind determines the partitioning methods based on the programmers' selection when it loads graph. The selection depends on the characteristics of graph algorithms, which will be introduced in the Chapter 3. GraphGrind also extends the EdgeMap function of Ligra with an automatic algorithm with 3-way graph layouts. GraphGrind can select corresponding traversal method with different graph layouts based on the size of frontier. The details will be introduced in the Chapter 4.

2.7 Further Related Work

METIS is a generic tool [51] to partition graphs by vertex or edge cut, do not produce good partitions for social network graphs. Moreover, they take much more time to compute than many graph algorithms. Sheep [64] is a distributed graph partitioner that produces high quality edge partitions an order of magnitude faster than METIS. The vertex cut is a greedy edge partitioning algorithm that minimises the number of cut vertices [33].

Bourse et al. [11] target distributed memory systems as it minimises the number of edges crossing partitions, which involve messages. This is not immediately relevant to the performance of shared memory systems. It is not immediately clear that the algorithm would perform well in the context of our system. The algorithm, moreover, approximates edge and vertex balancing. Experimental evaluation shows deviations in the vertex balance up to 50%, which would have

a prohibitively high impact on GraphGrind.

GraphChi [54] streams graph data from disk. It uses partitioning to obtain small vertex sets that fit in the main memory. It uses partitioning by destination with an equal number of edges per partition. The vertex data must be made to fit in memory by tuning the number of partitions. A large variation in vertices per partition implies that partitions with few vertices will leave a large portion of main memory unutilised in GraphChi.

X-Stream [80] uses what we call partitioning by source, but does not require edges to be pre-sorted. It aims for a uniform number of vertices per partition as it wants to keep only vertex data in fast memory (e.g., CPU cache), whereas edges are streamed in from slower memory (e.g., main memory). GraphX [34] is a Spark [107] library for graph analytics. It partitions edge lists using Spark's resilient distributed datasets (RDD) and supports user-defined partitioning schemes.

Agarwal et al. [4] study the execution of breadth-first-search on NUMA systems. They too organise the computation around work queues, spread over multiple sockets. They use efficient spinning locks and lock-free channels to synchronise threads and they introduce peephole optimisations, e.g., avoiding atomic operations by first checking if they will fail.

Cong et al. [16] analyse the memory locality of minimal spanning trees (MST) by collecting reuse distance profiles. They show that both temporal and spatial locality varies between algorithms for MST, and that parallel algorithms have worse locality. It is impossible to quantify if memory locality is in general good or bad on the basis of their study as they apply their analysis on a small graph. Memory locality refers to the property that memory accesses tend to refer to the same memory locations as recently executed memory accesses. It may be mea-

sured through working set size [20], or through cache miss rates, both approaches are equivalent [23, 84]. The memory hierarchy is utilised best when working sets are small compared to the sizes of the on-chip caches. When working sets are too large to fit in the caches, graph analytics become memory bound in previous research. Yuan *et al* [106] analytically model working set sizes for graph traversal. They introduce the notion of *vertex distance*, which captures the average number of verices accessed prior to re-visiting a vertex. They show that the vertex distance for BFS follows a geometric distribution in random graphs. In general, the vertex distance distribution is highly dependent on the degree distribution. This analysis demonstrates that locality varies with the shape and size of the graph, implying that larger graphs will have worse locality.

Hilbert space filling curves have been applied to optimise the iteration order of edge list traversal [66, 71]. Space filling curves improve memory locality for various algorithms [15, 21, 67]. They have experimented with Hilbert curves and have achieved speedup compared to CSC and CSR layouts when the number of partitions is sufficiently high and hardware atomics can be avoided.

Murray *et al.* [71] present a PageRank implementation with a COO layout and space filling curves. Their work uses graph partitioning to create parallelism. They consider partitioning-by-source and partitioning of the edge list after rearranging edges in Hilbert order. While the Hilbert order improves their performance by an order of magnitude, they have not considered partitioning-by-destination to improve locality and to ensure vertices are updated by a single thread. They also have no specific support for sparsely populated frontiers.

Frasca *et al.* [25] define an Adaptive Data Layout (ADL) to reorganise the graph after observing parallel access patterns on NUMA system. Dai *et al.* [18]

apply graph partitioning in the context of multi-FPGA systems. As such, locality may be optimised along the time dimension or along the spatial dimension. The authors optimise the time dimension due to the challenging nature of locality in the spatial dimension.

Most often, the graph partitioning problem is formulated as calculating a subset of the edges (or vertices) such that the number of edges crossing partitions is minimized [6, 24, 88, 92, 108]. Additionally, authors specify a constraint to balance the edges [50] or vertices [11, 33] in a partition.

The exact solution to the graph partitioning problem is NP complete (e.g., [24]). As such, many authors have considered approximate algorithms to achieve a close-to-optimal solution in polynomial time [6, 24]. These may produce partitions of similar quality as general-purpose graph partitioners such as METIS [51] in less time [24].

From a practical point of view, an important dimension in the problem space is whether one partitions the edge set or the vertex set. Partitioning the vertex set is more intuitive and leads to a problem of minimizing the edge cut, potentially under a constraint of edge balance [54, 80, 108]. Partitioning the edge set leads to better heuristics and higher-performing implementations [33]. Edges now belong to a partition, while vertices may be replicated. In this problem, rather than minimizing the edge cut, the optimization criterion is to minimize the amount of vertex replication, also known as vertex cut.

Streaming partitioning algorithms partition the graph in a single pass using a limited amount of storage [88, 92]. These algorithms compute approximations to the optimal partition by design. In practice, it has been shown that they can produce partitions of similar quality to METIS [51] in a fraction of the time [92].

Gonzalez *et al.* proposed *vertex cut*, a parallel streaming partitioning algorithm that minimizes vertex replication [33]. Li et al. [57] and Bourse et al. [11] proposed efficient edge-balanced partitioning methods. Bourse et al. [11] moreover investigate the interplay between edge balance and vertex balance, which is non-trivial if edge cuts are simultaneously minimized.

Some related work uses graph ordering to address performance issues during graph analytics. **Gorder** [98] proposes a general graph ordering method to improve the CPU computing, which provides an opportunity to apply the efficiency of graph algorithms without changing the implementation and data structures used. They design a general purpose approach to optimal permutation among all nodes by keeping vertices will be accessed frequently together locally, try to minimise the CPU cache miss ratio. George et al. [30] proposed the reverse Cuthill-McKee (**RCM**) algorithm , which is the well known graph ordering approach for reducing graph bandwidth. **SlashBurn** [58] is a graph ordering used in graph compression algorithm. It is exploiting the hubs and their neighbors to define an alternative community different from the traditional community. It is removing hubs from a graph creates many small disconnected components, and the remaining giant connected component is substantially smaller than the original graph. **LDG** [88] is a heuristic streaming partitioner for large distributed graph. **MultiGraph** [42] orders each active vertex and their corresponding outgoing edge to a thread for load balancing. But it requires that each tread identify its edge attributes. These graph reordering algorithms do not always work well in all situations, due to the NP-hardness of solving the exact problem. Intuitively, edge balance and edge cut minimization are partially contradictory constraints.

Compared to prior work, this dissertation optimises load balance and mem-

ory locality using a general purpose method based on the characteristics of graph algorithms, graph layouts and graph partitioning. In order to deal with load imbalance, GraphGrind introduces a classification of algorithms to distinguish whether they benefit from edge-balanced or vertex-balanced partitioning. Moreover, this dissertation introduces a simple graph ordering algorithm to balance the number of vertices and edges per partition together. For poor memory locality, this dissertation demonstrates that increasing the number of graph partitions is effective to improve temporal locality due to smaller working sets. In order to avoid vertex replication in graph partitioning, this dissertation designs an automatic graph traversal algorithm to switch between COO, CSR and CSC.

Chapter 3

Addressing Load Imbalance of Graph Partitioning

This chapter investigates how graph partitioning adversely affects the performance of load imbalance of graph analytics. It demonstrates that graph partitioning induces extra work during graph traversal and that graph partitions have markedly different connectivity than the original graph. Moreover, this chapter shows that the heuristic to balance CPU load between graph partitions by balancing the number of edges is inappropriate for a range of graph analyses. However, even when it is appropriate, it is sub-optimal due to the skewed degree distribution of social networks. Based on these observations, this chapter proposes GraphGrind [90], a new graph analytics system that addresses the limitations incurred by graph partitioning. It moreover proposes a NUMA-aware extension to the Cilk programming language and obtains a scale-free yet NUMA-aware parallel programming environment which underpins NUMA-aware scheduling in GraphGrind [90].

3.1 Introduction

Chapter 1 and Chapter 2 explain that graph partitioning has been proposed to isolate memory accesses to specific parts of the graph data. After partitioning, some vertices have zero in-degree or out-degree in some partitions. These zero-degree vertices results in an increase in memory usage across partitions. It is essential to remove these zero-degree vertices per partition to compress graphs. This compression can significantly reduce memory requirements and with its memory bandwidth. Shun *et al* [87] compress the destination IDs of vertices stored in the edge array of the CSR and CSC representations. They reduce memory usage up to 56%. It is orthogonal to the compressed representation of the CSC and CSR index arrays proposed in this work, as they pertain to edges only.

Moreover, Chapter 1 mentions that partitioning the edge set results in an imbalance in the number of vertices appearing in each partition. Alternatively, partitioning the vertex set results in an imbalance in the number of edges. There is significant load imbalance existing between partitions, either for loops iterating over vertices, or for loops iterating over edges. In order to deal with load imbalance problem of partitioning, this chapter classifies graph algorithms into ***vertex-oriented and edge-oriented*** based on the characteristics of graph algorithms.

- ***Vertex-oriented*** algorithms have nearly constant computation work per vertex. They use *vertex-balanced* partitioning ensures each partition has almost equal amount of vertices.

- ***Edge-oriented*** algorithms have nearly constant computation work per edge. They use *edge-balanced* partitioning ensures each partition has almost equal

amount of edges.

This chapter analyses heuristic graph partitioning in detail and identifies side effects that limit achievable performance. In particular, it shows that graph partitioning incurs an innate performance overhead, which stems from increased control flow and from the decreased connection density of the partitions. This chapter makes the following contributions:

- Section 3.2 analyses the characteristics of graph partitions and identifies how these limit performance.
- Section 3.3 analyses the characteristics of graph algorithms and classifies them into vertex-oriented and edge-oriented.
- Section 3.3 presents GraphGrind, a NUMA-aware graph analytics framework based on Ligra [85] and Polymer [108], that reduces the performance impact of graph partitioning. It has an improved graph representation, tuning the partitioning to the characteristics of the algorithm and improving the NUMA memory mapping of key data structures.
- Section 3.3 develops an extension to the Cilk parallel programming language [29, 46] that allows expression of NUMA affinity for parallel loops. It simplifies the design of GraphGrind and is generally applicable to enforce NUMA-aware scheduling in parallel programs.
- Evaluation in GraphGrind of Section 3.4 shows that it outperforms Ligra by 1.46x on average, and Polymer by 1.16x on average, on a variety of graph algorithms and datasets.

The remainder of this chapter is organised as follows. Section 3.2 presents the investigation through analysing the adverse impact of graph partitioning. Section 3.3 describes the design and implementation of GraphGrind. Section 3.4 presents an experimental evaluation of GraphGrind.

3.2 Motivation

3.2.1 Extra Work Induced by Partitioning

When partitioning the edge set, the list of edges of a vertex is split with parts of the list appearing in different partitions. As such, the edges for some vertices are stored in distinct partitions. Graph traversal must thus visit the vertex once for each replication. The additional cost of this is a small amount of control flow, lookups in the graph representation and checking whether the vertex is active. While these actions require only a few dozen assembly instructions, it is important to keep in mind that graph analytics perform little computation, typically less than a dozen assembly instructions per edge. Moreover, the overhead involves several main memory accesses as these algorithms are memory intensive.

Figure 1.1 shows the average replication factor of vertices for various degrees of partitioning. The graphs are described in Table 2.2. It shows data for 6 of the 8 graphs as the remaining 2 behave similarly. Graphs with few edges per vertex (USARoad and Friendster) have the lowest replication factors while highly skewed graphs (Twitter and Orkut) have the highest. Assuming 4 partitions, replication factors are often in the range 2–3, which implies that the control flow overhead of graph traversal is repeated 2 to 3 times. Our measurements show that the vertex

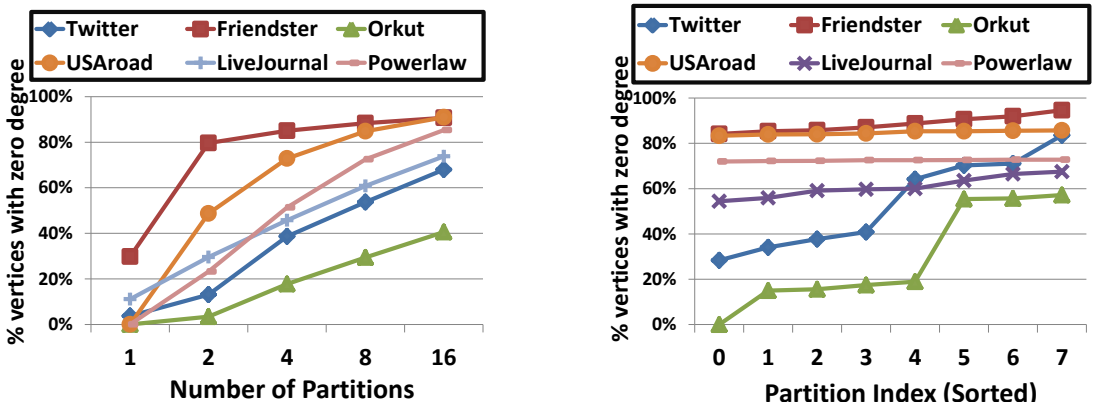


Figure 3.1: Percentage of vertices with zero out-degree averaged across all partitions (left) and variation across each of 8 partitions (right).

replication results in an instruction count increase of up to 18%.

Figure 1.1 moreover shows that the graph partitioning algorithm studied in this chapter achieves a comparable replication factor for the Twitter graph as the more elaborate algorithm in [33]. It may thus assume that the conclusions of this chapter are independent of the partitioning algorithm used, as the conclusions build on the observation that the replication factor is larger than one.

3.2.2 Sparsity of Graph Partitions

If vertices are not replicated across all partitions, then by necessity vertices will not have incoming or outgoing edges in several of the partitions. Figure 3.1 (left) shows the average number of vertices with zero degree for varying degrees of partitioning by destination. Similar results hold for partitioning by source. The fraction of vertices with zero outgoing edges shoots up quickly as more partitions are introduced, exceeding in many cases 50% for 4 partitions. Moreover, real-world social networks have strongly imbalanced partitions (Figure 3.1 (right)). In contrast, the partitions of synthetic graphs, intended to model real-world graphs,

have equal numbers of unconnected vertices in each partition. Interestingly, the Friendster graph has fairly equal partitions. The sparsity of graph partitions leads to an opportunity: if the absent vertices in a partition can be avoided iterating over, then the instruction count increase for these vertices can be restricted only to the partitions where the vertex occurs. To this end, GraphGrind uses a variation of the CSR representation where zero-degree vertices are not recorded.

3.2.3 Balancing Edges vs. Vertices

It is hard to partition a social network graph in a balanced way due to its skewed degree distribution. Social network graphs like Twitter and Friendster have highly different numbers of vertices per partition when balancing the number of edges.

The imbalance of the number of vertices per partition has an important impact on performance. First, many graph algorithms make passes over vertices apart from passes over the edges. As such, the work performed per graph partition is not only proportional to the number of edges, but also depends on the number of vertices.

Secondly, not all algorithms perform a fixed amount of work per edge. Instead, algorithms such as BFS, betweenness-centrality, Bellman-Ford and K-Core visit at most one active edge per active vertex. For them, balancing the edges between partitions does not result in a balanced CPU load.

Thirdly, an imbalance in the number of vertices per partition results in a skewed utilisation of memory and creates hotspots for certain partitions. This unnecessarily drives to scale-out distributed systems to higher degrees of parallelism to drive the worst-case partition size down, even if the computation does

not warrant scaling out. In shared memory systems the memory imbalance may be combated by storing data in a sub-optimal NUMA node, which results in the lesser evil of remote NUMA accesses.

Increasing the number of partitions may seem to avoid skewed partitions. This is however not true. Based on the measurements, the presence of highly-connected vertices remains an issue with higher degrees of partitioning as some partitions have twice as many vertices as others. We conclude that the graph partitioning needs to balance CPU load and should be adapted to characteristics of the algorithm.

Hence, how to deal with the load imbalance partitioning becomes an important issue in the graph analytics. This dissertation aims to provide a general method to deal with load imbalance of graph analytics. Chapter 2 presents our solution design, GraphGrind, which implements all the optimisations behind an abstract, which is simple for programmers to use directly. GraphGrind extends Ligra with graph partitioning to speedup graph analytics using Cilk as parallelism tool. In the first stage design of GraphGrind, it distinguishes graph algorithms into vertex-oriented and edge-oriented. Different algorithm uses corresponding partitioning method to balance CPU load. For instance, edge-oriented algorithms perform the work is proportional to the number of edges. Hence, edge-oriented algorithms need to use edge-balance partitioning to ensure each partition has almost same number of edges to balance the work per partition and CPU load.

3.3 GraphGrind: Design and Implementation

A extension programming interface with a *cache* is used for ***backward*** *edge-map* traversals. While *edge-map* may execute in parallel, it traverses the incoming edges of a vertex sequentially when the number of vertices is not very large (less than 1000). Compilers should, in principle, be able to hold the intermediate updates for the destination vertex's value in registers. However, the complexity of control flow and pointer aliasing prohibits this in practice. GraphGrind allows the programmer to specify how to cache intermediate updates for the function Fn. This explicit notation allows compilers to allocate them to registers and involves a cache type definition and 3 functions to initialise the cache, to update it and to commit it to the main state.

Listing 3.1 shows the Fn operator for PageRank. The *update*() function is used when the system knows that only one thread will update a vertex. This is typically during a *backward* traversal. *updateAtomic*() is used to rule out data races during *forward* traversal. And the *cache* is used for cache intermediate sequential updating. First, the *create_cache*() creates a cache contains the values of destination vertex. And then it will update the value to cache values from source vertex. Finally, the *commit_cache* will pass the value to destination vertex.

3.3.1 Frontier Representation

This part adapts the representation of frontiers between bitmaps and arrays of vertex IDs on-the-fly, depending on their density [43]. Frontiers are created either by constructors, or by the *edge-map* and *vertex-map* functions. From the users point of view, frontiers are immutable. One of the constructors creates a

Listing 3.1: Edge operator for PageRank

```
1 template <class vertex>
2 struct PR_Fn{
3         double* p_curr, *p_next;
4         vertex* V;
5         static const bool use_cache = true; // cache is implemented
6         struct cache_t { double p_next; };
7         PR_F(double* _p_curr, double* _p_next, vertex* _V) :
8         p_curr(_p_curr), p_next(_p_next), V(_V) {}
9         inline bool update(intT s, intT d) {      // Backward
10                p_next[d] += p_curr[s]/V[s].getOutDegree();
11                return true;
12        }
13        inline void create_cache(cache_t &cache, intT d){
14                cache.p_next = p_next[d];
15        }
16        inline bool update(cache_t &cache, intT s) { // Backward opt.
17                cache.p_next += p_curr[s]/V[s].getOutDegree();
18                return true;
19        }
20        inline void commit_cache(cache_t &cache, intT d) {
21                p_next[d] = cache.p_next;
22        }
23        inline bool updateAtomic (intT s, intT d) { // Forward
24                writeAdd(&p_next[d],p_curr[s]/V[s].getOutDegree());
25                return true;
26        }
27        inline bool cond (intT d) { return true; }
28 };
```

frontier containing all vertices. It is explicitly recorded this property in the frontier to omit checks of the frontier and speed up graph traversal. Remember that graph analytics typically perform little work per edge. As such, any reduction in instruction count has a measurable impact.

This optimisation affects traversal with dense frontiers. The backward traversal benefits much more from this optimisation as it performs more lookups in the frontier, namely once per edge vs. once per vertex in the case of the forward traversal. It similarly optimises the *vertex-map* operation and any auxiliary loop

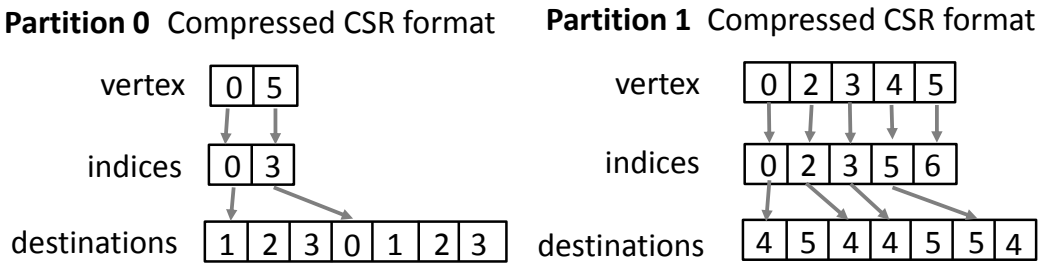


Figure 3.2: Compressed CSR format.

iterating over the frontier.

3.3.2 Compressed Graph Representation

This section modifies the CSR and CSC representation to combat efficiency issues with zero-degree vertices. Compress the index array by storing only information for vertices with non-zero degree and store the vertex ID with it. Figure 3.2 shows the modified CSR format for the graph partitions of Figure 2.4. In Partition 0, vertex 0 and 5 have out-edges which are home to vertex 0, 1, 2 and 3. In Partition 1, only vertex 1 has zero degree, so it is not stored. The representation reduces the size of the index array due to the high number of zero-degree vertices. The main benefit, however, is that a sequential edge traversal becomes more efficient as iteration over the index array automatically skips all zero degree vertices.

GraphGrind stores each graph partition in the CSR and CSC representations in order to support the direction-reversing technique. I.e., a *dense forward* traversal uses CSR while a *dense backward traversal* uses CSC. This representation is, however, not efficient for traversals with sparse frontiers as these are dominated by control flow, which is aggravated by the replication of vertices. As such, GraphGrind stores a *non-partitioned* copy of the original CSR representation of the graph specifically for sparse traversals. As such, it stores three copies of the

graph for undirected graphs, and two copies for directed graphs (as the CSR and CSC representations are equal for directed graphs).

3.3.3 Partition Balancing Criterion

It is argued above that balancing the number of edges across partitions does not necessarily result in the best balancing of CPU time. Instead, some algorithms observe better CPU load balancing when the number of vertices in each partition is about equal. GraphGrind adds a parameter to the algorithm specification that shows its preference for a balanced edge partitioning vs. a balanced vertex partitioning. This parameter is checked during graph ingress in order to select the balancing criterion for graph partitioning. Our balanced vertex partitioning is similar to Algorithm 2, except that we strive for $|V|/P$ destination vertices in each partition.

Balancing vertices is appropriate for 3 of the 8 algorithms that we use in the experimental evaluation. The algorithms are commonly used in prior work. As such, this property is sufficiently important to ask programmers to record it. The property is easily derived from the algorithm specification.

3.3.4 NUMA Optimisation

Common data placement strategies are to allocate data in a specific NUMA node or to distribute the data across nodes. Thread placement is optimised such that the thread has a low latency/high bandwidth connection to the NUMA domain holding its most frequently accessed data. This two-pronged strategy allows for many optimisations, such as co-locating threads with data and spreading data

Table 3.1: NUMA allocation and binding strategy

Data structure	**NUMA allocation**
full graph	interleaved
graph partition	allocate on one node
vertex arrays	match home partition
Operation	**NUMA binding**
edge-map (sparse)	none
edge-map (dense)	bind to holding node
vertex-oriented loops (e.g., vertex-map)	equally distribute loop iterations over NUMA nodes

and threads across NUMA domains to enhance memory bandwidth.

Graph partitions can enforce NUMA-local access as each partition can be stored and processed within the confines of one NUMA node. Prior work has advocated to replicate frontiers and algorithm-specific data arrays on each NUMA node [108]. Accordingly, memory accesses are NUMA-local, except when interchanging data across nodes.

GraphGrind follows a different route, which is summarised in Table 3.1. The full graph is stored in an interleaved fashion over the NUMA nodes. As the full graph is used with sparsely populated frontiers only, the memory accesses are few and hard to schedule optimally. The size of sparse frontier is usually small, e.g., only one or two active vertices of BFS. Small size of frontier leads to the memory accesses are few and random. The special allocation of whole graph may be lead to many unnecessary waste during sparse iteration. For instance, during the sparse iteration with special partitioned allocation, there is only one active vertex, however, the other special allocation in the other NUMA nodes also need to be traversed, even there is no work. Hence, interleaved allocation provides a good compromise for sparse iteration.

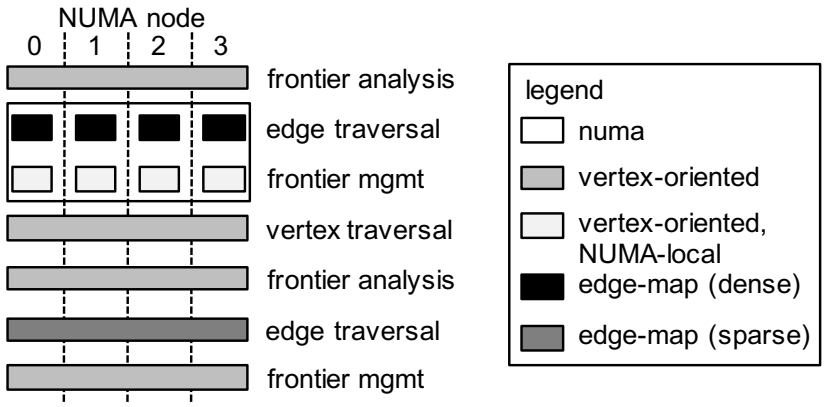


Figure 3.3: Schematic of loop structures and their NUMA-aware scheduling.

Figure 3.3 shows that graph partitions are spread over NUMA nodes in such a way that each partition is stored on one NUMA node and all NUMA nodes hold the same number of partitions. Algorithm 3 shows that a graph traversal over a partition is scheduled on the NUMA node that holds that partition. This ensures that the majority of memory accesses are issued against the local NUMA node.

Figure 3.3 also presents that we distribute *vertex arrays* over NUMA nodes, storing the element for each vertex on the same NUMA node as its home partition. As such, the *edge-map* operation that is *writing* data to a vertex element performs NUMA-local accesses. This placement incurs some *false sharing*, as NUMA placement works on the granularity of virtual memory pages. As such, a small fraction of the vertices will be placed on a remote NUMA node. E.g., assuming 1 M vertices spread over 4K pages and 4 bytes/vertex property, it will have some vertices placed in the wrong NUMA node. There may be 3 pages partially on the wrong NUMA node, or crossing NUMA node boundaries, so at most 3*250 vertices are placed sub-optimally, which is about 750/1M , at most 1 in

Algorithm 3 2-way partitioned GraphGrind edgemap

input : Graph $G = (V, E)$; frontier F; function Fn; bool Fwd
output : New frontier containing updated vertices
side effect: Fn applied to $(u, v) \in E : u \in F$

```
11 if #{(u,v) ∈ E : u ∈ F} > threshold then                  // Dense frontier
12     F_new = {} if Fwd then
13         for G_p : partitions of G do                        // NUMA-aware
14             edgeMapDenseForward(G_p, F, Fn, F_new)
15     else
16         for G_p : partitions of G do                        // NUMA-aware
17             edgeMapDenseBackward(G_p, F, Fn, F_new)
18     end
19     return F_new
20 else
                                                             // Sparse frontier
       return edgeMapSparse(G, F, Fn)
21 end
```

1000 will be stored in a different node. The distribution of vertex arrays may be highly skewed due to the imbalance of vertices in each partition. Loops iterating over the vertex arrays, such as *vertex-map* and loops that analyse frontiers, are however scheduled such that the loop iterations are equally spread across NUMA nodes. While this induces some remote NUMA accesses, it is far more important to load-balance these loops than it is to optimise NUMA-awareness.

An alternative strategy is to replicate the vertex arrays on each NUMA node [108]. It is sub-optimal due to the additional memory traffic that is required to replicate and to merge vertex arrays. In contrast, our NUMA placement and scheduling rules guarantee that an *edge-map* operation on a graph partition only writes to vertex array elements stored on the local NUMA node. Read operations may be remote, but these have lower impact on performance. As such, we obtain good NUMA locality without incurring the overhead of replicating data.

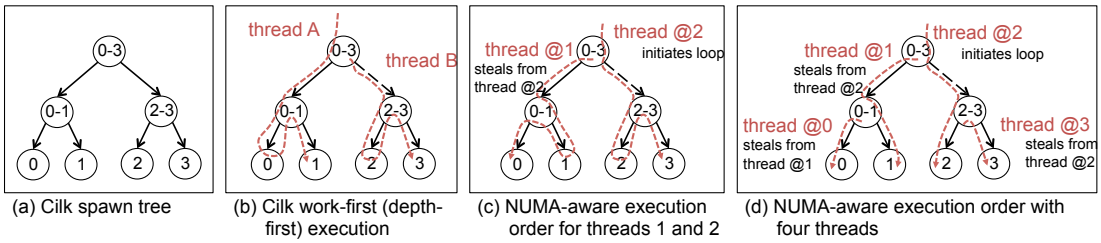


Figure 3.4: NUMA-aware work-stealing. "Thread @n" represents any thread executing on socket "n".

3.3.5 A NUMA-Aware Cilk Extension

GraphGrind is built on Cilk [29]. The Cilk language and runtime system have been modified to support NUMA-aware scheduling and work stealing. A minimalistic modification has been deliberately searched for as to not affect space- and time-efficiency [10] and implement this in Intel Cilkplus version 1.2 [46]. A proof of the space and time bounds to future work is delegated.

GraphGrind extends the programming language with a pragma "`#pragma cilk numa(strict)`" that can be supplied immediately preceding a `cilk_for` loop, similarly to the existing *grainsize* pragma. The programmers do not need to know how to do NUMA-aware design for Cilk tool. They can use the edgeMap and vertexMap function directly with NUMA-aware design. The NUMA pragma indicates that loop iteration i should preferably be executed on cores associated to NUMA domain i. The assumption that the number of loop iterations does not exceed the number of NUMA domains is a pragmatic one. Programmers may split loops over a NUMA-aware outer loop and a normal **cilk_for** innner loop that executes only on the NUMA domain encoded in its calling context.

GraphGrind also provides a NUMA-aware helper function that changes the execution order of loop iterations based on Figure 3.4 (b). The thread that

executes an instance of the helper function checks its current NUMA domain and first executes the sub-range that matches its NUMA domain. E.g., if a thread on NUMA domain 2 initiates execution of the loop, it executes the range 2-3 before the range 0-1 (Figure 3.4 (c)). This strategy is applied recursively: a thread on NUMA domain 3 will first execute loop iteration 3. This way, work is distributed to the correct NUMA domain with a minimal work stealing (Figure 3.4 (d)).

Work stealing is modified to respect the NUMA constraints. Every dynamic function call is marked by the helper function with the range of NUMA nodes where the function may execute. This range reflects the iteration sub-range of the loop. The range is copied over to recursively called functions. A worker that selects a victim thread inspects the NUMA range of the victim's oldest function and aborts the work stealing attempt if the NUMA range does not contain its own NUMA node. By default, NUMA ranges are not set and work stealing proceeds as normal.

The algorithm is robust against anomalous conditions such as absence of active threads on a NUMA domain and a mismatch between the number of NUMA domains specified by the program and those in hardware. In both cases, pending iterations are executed on sub-optimal NUMA domains. The NUMA extension supports non-commuting reductions [27] and pedigrees [55]. Both constructs depend on the execution order of function calls, which the helper function disrupts.

3.4 Experimental Evaluation

GraphGrind is evaluated on a 4-socket 2.6GHz Intel Xeon E7-4860 v2, totalling 96 threads, with 256 GB of DRAM. It is compiled all codes using a modified

version of the Clang compiler which implements the NUMA extension to Intel Cilkplus [46]. Evaluation uses 8 graph algorithms (see Table 2.1) using 8 widely used graph data sets (see Table 2.2). All results are averaged over 10 executions.

3.4.1 Performance Comparison

The performance of GraphGrind is compared against leading graph analytics systems for shared-memory, namely Ligra[1] [85] and Polymer [2] [108] (Table 3.2). It limits the comparison to these systems as it has been established [108] that Polymer out-performs Galois [77] and X-stream [80]. GraphGrind and Polymer both use 4 partitions to match the NUMA characteristics of our hardware. All systems use 96 threads. The backward PageRank algorithm is shown for Polymer as the forward version, presented in [108], contains errors. The absolute execution times depend on our hardware, compiler version and randomly generated graphs. Moreover, some algorithms are sensitive to the start vertex, which in this experiments is vertex 100 for all graphs, e.g., BFS, BC and BF. These algorithms aim to find a tree of the graph structure, which needs a root (start) vertex in the beginning. They will traverse different path of the graph using different start vertex. Here, we use 100 as a general parameter. The reported trends match previously reported results. Overall, GraphGrind outperforms Polymer and Ligra for all algorithms and all graphs. In a few cases, GraphGrind performs on par with other systems. These are labeled in bold-face as well.

Table 3.2 shows the average time of Ligra, Polymer and GraphGrind running 10 rounds. Table 3.3 shows the deviation values of Ligra, Polymer and Graph-

[1] `https://github.com/jshun/ligra.git`
[2] `http://ipads.se.sjtu.edu.cn:1312/opensource/polymer.git`

Table 3.2: Runtime in seconds of GraphGrind, Polymer, Ligra. The fastest results are indicated in bold-face. Execution times that differ by less than 1% are both labeled. Missing results occur as not all systems implement each algorithm. GraphGrind and Polymer use 4 partitions.

Algo.	Graph	GG	Polymer	Ligra	Algo.	Graph	GG	Polymer	Ligra
CC	Twitter	**1.810**	2.580	2.878	PR-Delta	Twitter	**20.560**	24.120	29.890
	Friendster	**5.924**	8.030	7.330		Friendster	**36.097**	36.600	62.100
	Orkut	**0.122**	0.180	0.138		Orkut	**1.244**	1.310	3.472
	LiveJournal	**0.111**	0.177	0.125		LiveJournal	**1.013**	1.110	1.138
	Yahoo_mem	**0.042**	0.049	0.063		Yahoo_mem	**0.831**	1.094	1.640
	USAroad	**35.348**	36.730	38.910		USAroad	**2.124**	2.260	2.905
	Powerlaw	**1.168**	2.110	1.680		Powerlaw	**10.659**	14.100	16.900
	RMAT27	**2.305**	3.220	2.444		RMAT27	**8.645**	12.120	14.500
BC	Twitter	**1.771**		4.130	SPMV	Twitter	**2.251**	2.860	4.610
	Friendster	**3.394**		5.490		Friendster	**3.624**	5.220	9.010
	Orkut	**0.149**		0.160		Orkut	**0.148**	0.208	0.630
	LiveJournal	**0.197**		0.334		LiveJournal	**0.060**	0.096	0.151
	Yahoo_mem	**0.091**		0.110		Yahoo_mem	**0.033**	0.045	0.063
	USAroad	**4.402**		5.174		USAroad	**0.077**	0.128	0.166
	Powerlaw	**2.118**		2.300		Powerlaw	**0.655**	**0.661**	0.707
	RMAT27	**2.073**		2.360		RMAT27	**1.963**	2.210	2.830
PR	Twitter	**15.979**	20.400	23.660	BF	Twitter	**1.489**	1.618	2.213
	Friendster	**38.249**	41.8	43.300 0		Friendster	**6.498**	7.193	7.690
	Orkut	**1.596**	1.660	2.240		Orkut	**0.213**	0.310	0.354
	LiveJournal	**0.652**	0.688	0.708		LiveJournal	**0.258**	0.293	0.284
	Yahoo_mem	**0.234**	0.262	0.278		Yahoo_mem	**0.146**	0.200	0.173
	USAroad	**0.933**	1.220	1.582		USAroad	**21.992**	24.110	26.310
	Powerlaw	**10.394**	12.716	13.600		Powerlaw	**10.326**	11.112	12.600
	RMAT27	**17.517**	23.21	28.600		RMAT27	**1.665**	1.933	2.180
BFS	Twitter	**0.254**	0.298	0.319	BP	Twitter	**38.896**	**38.900**	56.980
	Friendster	**0.896**	**0.899**	1.210		Friendster	**58.704**	66.210	129.000
	Orkut	**0.039**	0.043	0.044		Orkut	**2.223**	3.110	5.538
	LiveJournal	**0.050**	0.068	0.078		LiveJournal	**1.026**	1.420	1.940
	Yahoo_mem	**0.025**	0.026	0.033		Yahoo_mem	**0.448**	0.455	1.124
	USAroad	**1.750**	1.855	2.009		USAroad	**1.024**	1.660	1.462
	Powerlaw	**0.595**	0.601	**0.599**		Powerlaw	**15.264**	15.530	19.500
	RMAT27	**0.412**	0.421	0.429		RMAT27	**32.994**	43.320	58.230

Table 3.3: The standard deviation of 10 rounds of BFS and PageRank using Twitter graph and Friendster graph.

	Twitter			Friendster		
Algorithm	Ligra	Polymer	GG	Ligra	Polymer	GG
BFS	0.007	0.009	0.004	0.020	0.022	0.015
PageRank	0.073	0.056	0.052	0.093	0.076	0.063

Grind with 10 rounds. Because all the algorithms and graphs show a similar trend, here, we use one vertex-oriented algorithm (BFS) and one edge-oriented algorithm (PageRank) with Twitter graph and Friendster graph as analysis example. Of course, the mean value of each framework is statistically different. All deviation values are small, and GraphGrind has the smallest value. It shows that the execution time of these three frameworks becomes stable during 10 rounds. And GraphGrind has faster execution time compared to Ligra and Polymer per round. Hence, this dissertation uses average time as a stable and fair value to do comparison.

These standard deviation values of 10 rounds are small, which shows that there is a stable execution time using 10 rounds. And we compared each round time, we found for each iteration, GraphGrind out-performs Ligra and Polymer. Hence, this dissertation uses average time as comparison.

The performance improvements are significant: up to 326% faster than Ligra (SPMV with Orkut graph) and up to 82.2% faster than Polymer (BP with USAroad graph). The smallest speedup appear for BFS, as there is already little computation going on. The superior performance of GraphGrind results from a combination of optimisations.

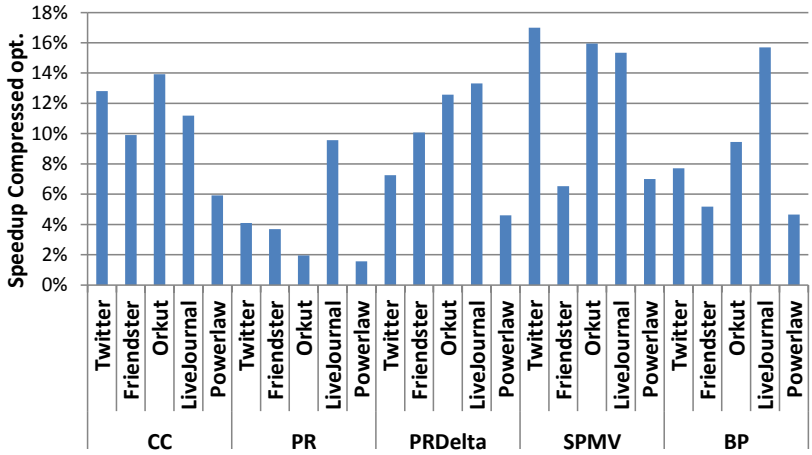


Figure 3.5: Speedup of compressed graph compared to visit zero-degree vertices.

3.4.2 Compressed Graph Representation

GraphGrind's graph data structure prunes vertices with zero degree from the representation. It will be presented later that this saves significant spaces compared to the CSC and CSR representations used by Polymer. Moreover, by not storing these vertices, *edge-map* traversals no longer need to visit them. Figure 3.5 shows the speedup resulting from the graph representation for 5 algorithms, which ranges between 2% and 16%. Twitter and LiveJournal benefit most due to the high sparsity of graph partitions.

3.4.3 Adapting Graph Partitioning

GraphGrind removes CPU load imbalance through selecting an appropriate criterion to balance the graph partitions. We identified 3 of the evaluated algorithms (BFS, BC and BF) prefer an equal number of vertices in each partition. And, the BFS and BC have the ability to skip the other active edges of one vertex if one active edge has been traversed and updated during the traversal. They do not

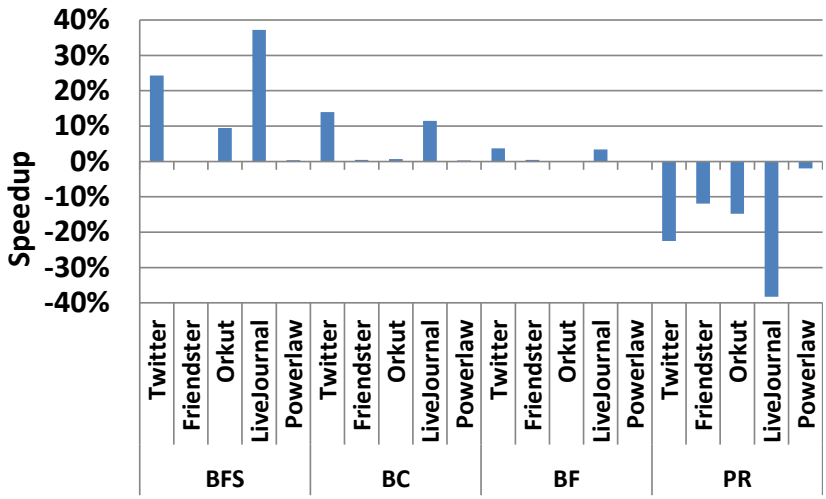


Figure 3.6: Speedup of balancing vertices compared to balancing edges in graph partitions.

need to traverse all active edges anymore. The others prefer a uniform number of edges. Figure 3.6 shows the speedup obtained by balancing vertices over balancing edges for these 3 algorithms and PR. Results for a subset of the graphs are shown, the remaining graphs behave similar to the ones shown. The partitioning has negligible impact for Friendster and PowerGraph, which have a balanced number of vertices per partition in either case (see Figure 3.1). Graphs with unbalanced partitions see important improvements with vertex-balanced partitions, with up to 37% speedup for LiveJournal.

Vertex-balanced partitioning is appropriate only for algorithms with fixed amount of work per vertex. Other algorithms, like PR, have a strong preference for edge-balanced partitioning. The conclusion shows that it is crucial to balance partitions appropriately to the algorithm. A parameter is needed in the command-line to switch between vertex-oriented algorithm and edge-oriented algorithm.

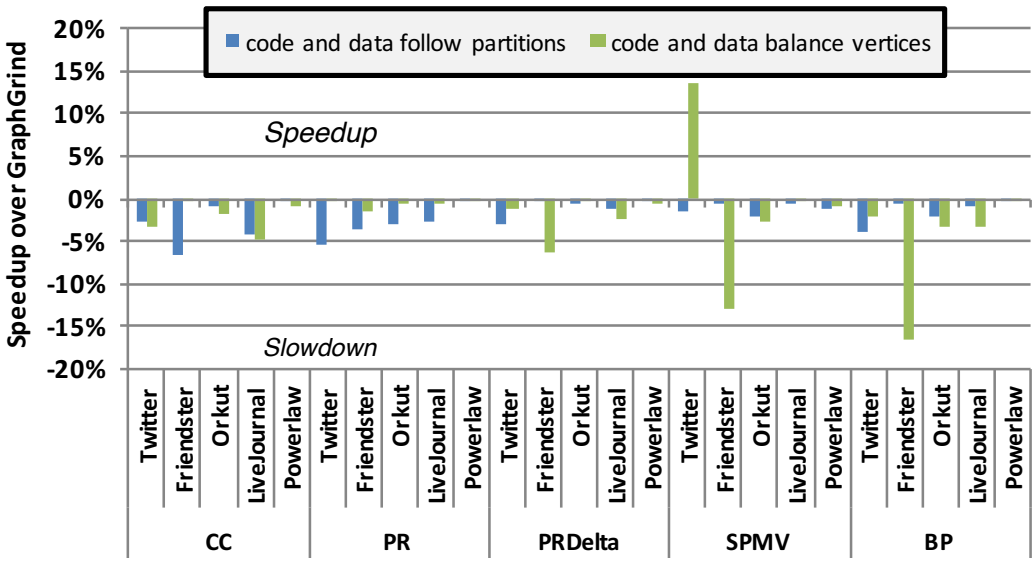


Figure 3.7: Impact of NUMA decisions for vertex arrays. GraphGrind may be described as data follow partitions, code balances iterations.

3.4.4 NUMA Optimisation

Various choices can be made for the placement of vertex arrays, i.e., arrays storing frontiers or per-vertex application-specific data. GraphGrind places the vertex arrays such that each vertex is co-located with its home partition. Vertex-oriented loops, such as those in *vertex-map*, are typically short and have well-balanced work per iteration. As such, GraphGrind distributes the iterations equally across threads, even though this results in remote NUMA accesses.

It compares two variations on the NUMA policy (Figure 3.7): (i) placing vertex data and scheduling iterations on their home partition; (ii) equally spreading vertex data and iterations across all NUMA nodes. Option (i) aims to avoid remote NUMA access during vertex-oriented loops. This is however uniformly worse than GraphGrind's policy. It shows that CPU load balance is simply more important than NUMA locality for the vertex-oriented loops.

Option (ii) load-balances vertex-oriented loops and tries to minimise remote NUMA accesses by spreading vertex arrays to match the distribution of iterations. This results in worse performance in nearly all cases as the placement decision is sub-optimal for the *edge-map* operator. This operator performs the majority of main memory accesses and will incur excess remote memory accesses when vertices are not co-located with their home partition.

An interesting effect occurs when SPMV processes the Twitter graph, as in this case an increase in remote memory accesses during *edge-map* results in improved performance. GraphGrind contrasts this against Friendster, where the same effect results in performance degradation. We measured the local and remote memory accesses incurred and observe that both GraphGrind and option (ii) incur the same total number of memory accesses and that option (ii) incurs an increased number of remote accesses for both graphs.

The performance difference between the graphs, however, results as Twitter has highly skewed partitions: The number of elements of vertex arrays accessed on one NUMA node is much higher than on other NUMA nodes. Where GraphGrind directs those accesses to the local NUMA node, option (ii) spreads them across nodes. This way, option (ii) can share the unused memory bandwidth on one NUMA node with the computation on another node. On Friendster, GraphGrind is faster than option (ii) because Friendster has relatively uniform partitions and performs more memory accesses per unit of time. As such, all NUMA nodes are equally stressed and there is no benefit in making remote accesses.

These results show that a careful trade-off is required to optimise NUMA placement, as option (i) incurs fewer remote memory accesses than GraphGrind, yet has worse performance. In rare cases can remote accesses result in perfor-

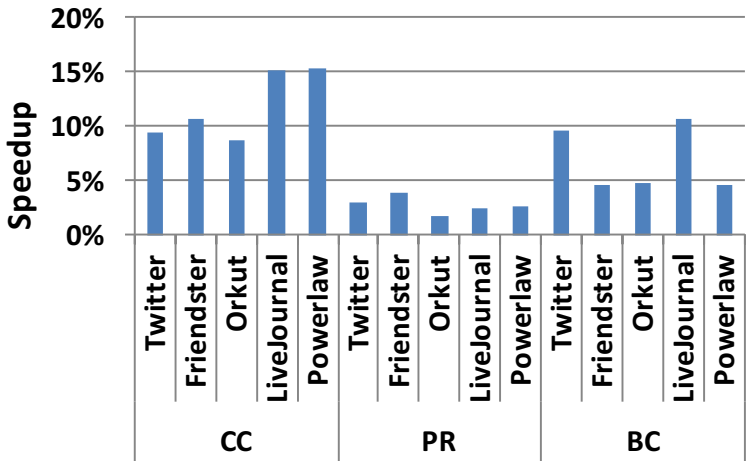


Figure 3.8: Speedup due to holding intermediate values in registers.

mance improvement due to imbalance in memory traffic.

3.4.5 Peephole Optimisations

GraphGrind marks frontiers that are initialised to contain all vertices such that an optimised *edge-map* can avoid memory accesses and control flow related to frontier access. Only algorithms that initialise frontiers this way can benefit. The algorithms using backward traversal (CC and PR) benefit most, up to 8%, as the backward traversal queries the frontier once for every edge, while the forward traversal queries it only once per vertex. The speedup is modest, but consistently positive. It moreover requires no user intervention.

GraphGrind allows programmers to define a cache, which allows the compiler to store intermediate values in registers (the cache) and avoid memory accesses. This optimisation is relevant only during backward traversal. Figure 3.8 shows that for the relevant algorithms, cache results in a speedup between 2% and 15%.

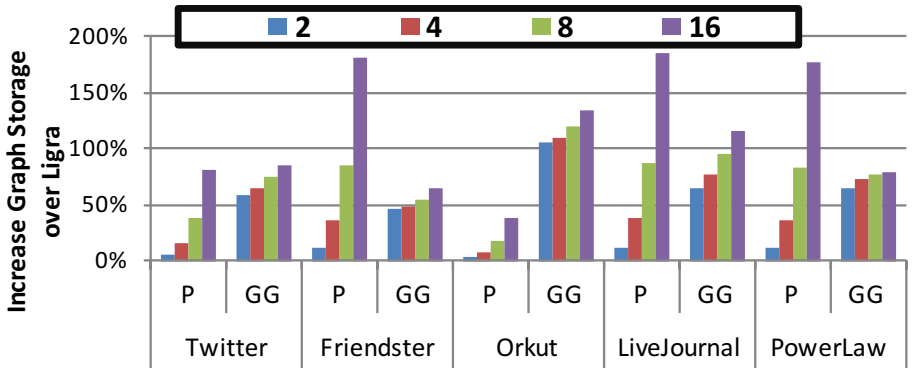


Figure 3.9: Increase of graph storage for Polymer (P) and GraphGrind (GG) compared to Ligra.

3.4.6 Memory Usage

Figure 3.9 shows the additional memory used on graph data for Polymer and GraphGrind compared to Ligra. Polymer stores each graph partition in CSR and CSC format (as in Ligra) using index arrays of length $|V|$. Because of this, the memory consumption of Polymer grows as $P|V|$ for P partitions. As GraphGrind stores only vertices with non-zero degree in the index arrays, its memory usage grows more slowly and follows the vertex replication factor (Figure 1.1). However, as GraphGrind stores an additional copy of the graph for sparse traversal, it starts at a 50% increase compared to Ligra for directed graphs. Overall, GraphGrind's memory consumption is more scalable than Polymer's.

3.5 Summary

This chapter studies graph partitioning in the context of NUMA-aware data placement and code scheduling. It analyses the performance issues that graph partitioning inadvertently introduces, including load imbalance, increased work per vertex, and a significantly reduced connection density. Combined, these problems

imply that graph partitioning is inherently unscalable to large partition counts.

This chapter proposes several techniques to counter-act the identified performance issues and implements these in GraphGrind. Finally, GraphGrind achieves significant speedup compared to prior work, outperforming Polymer, the most recent contender, by 1.16x on average. This chapter moreover shows that fully minimising remote memory accesses is not optimal in irregular computations. Instead, one needs to strike a careful trade-off between remote accesses and CPU load balancing.

Chapter 4

Enhancing Memory Locality with Graph Partitioning

Poor locality has been repeatedly recognised as an important issue [9, 16, 61, 106, 110], yet few authors have proposed solutions to *temporal locality*. This chapter investigates how to improve *temporal locality* using graph partitioning. However, realising performance improvement through graph partitioning poses several challenges and requires rethinking the classification of graph algorithms and preferred data structures. This chapter introduces the notion of *medium-dense* frontiers, a type of frontier that is sufficiently dense for a bitmap representation, yet benefits from an indexed graph layout. Using three types of frontiers, and three graph layout schemes optimised to each frontier type, it proposes an edge traversal algorithm that autonomously decides which type to use. The distinction of forward vs. backward graph traversal folds into this decision and need no longer be specified by the programmer.

4.1 Introduction

Section 1.1.2 presents that graph analytics operations compute a value associated to the edges or vertices of a graph by iteratively updating the values until convergence, or a fixed number of iterations have been performed [59,63]. Values are updated as a function of the values on connected vertices. As such, each update requires to read and/or write values associated to the end-points of edges, which may be randomly dispersed through memory depending on the structure of the graph. Memory locality of graph analytics is thus highly data-dependent.

For NUMA-aware optimisation, Chapter 3 has use partitions as they direct updates of values to specific subsets of the vertices. This chapter will show how the same partitioning technique can be used to control temporal locality. For such a partitioning scheme, this chapter demonstrates that *reuse distances* are reduced as the graph is more finely partitioned.

This chapter furthermore investigates how far GraphGrind can scale graph partitioning in order to maximally benefit from temporal locality. Prior work has acted to match the number of graph partitions to the number of nodes in a distributed (scale-out) system [33], or to the number of NUMA nodes in a scale-up system [108], such as X-stream [80] and GraphChi [54], the partition the graph such that a partition fits in memory. Out-of-core systems determine the partitioning factor such that individual partitions fit in core memory [38, 54, 80]. However, this dissertation targets scale-up systems and identifies multiple good reasons to strive for high degrees of partitioning:

1. Improving temporal locality of memory accesses;

2. Confining all updates to a value to one partition and one thread, which

boosts performance by avoiding hardware atomic operations;

3. Load balancing work across threads as the amount of work per partition varies with graph structure and active edge set.

Implementing a large number of partitions, however, requires careful design of the graph data structures. For COO representation, all edges must be traversed during each iteration of the graph algorithm, which is prohibitively expensive when few edges are active [4, 43, 108].

Representations such as CSR and CSC effectively provide an index into the edge list, allowing efficient lookup of the edges incident to active vertices. This representation, however, does not scale to a large number of partitions due to either edges or vertices crossing partitions, requiring replication of those edges or vertices in all relevant partitions [33]. This chapter develops a composite graph storage scheme that combines CSC/CSR with COO in the best possible way to enable a large number of graph partitions. A new graph storage scheme of GraphGrind moreover adapts to properties of the graph algorithm.

The remainder of this chapter is organised as follows. Section 4.2 discusses graph partitioning algorithm and its ensuing properties. Section 4.3 presents the rationale and design of GraphGrind. Section 4.4 reports on the experimental evaluation of graph partitioning and shows how it improves performance.

4.2 Graph Partitioning

This section defines target graph partitioning approach and presents how it improves temporal locality.

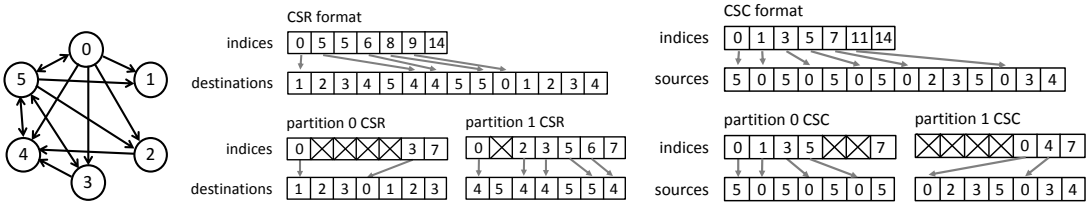


Figure 4.1: Graph layout in CSR and CSC formats and the corresponding graph partitions when partitioning by destination.

4.2.1 Locality of Partitioning

The example graph of Figure 4.1 helps to understand why locality improves temporal locality. The graph has 6 vertices and 14 edges. The CSR and CSC layouts of the whole graph are shown at the top of the figure. The same layouts are shown below when the graph is partitioned by destination.

When traversing the graph, it consults two types of data for each edge: For the source vertex it consults *current* data which includes a current frontier stored in bitmap format as well as an application-specific array storing per-vertex data. For the destination vertex it consults *next* data which includes the next frontier and the next version of the application-specific data. In the remainder, it will simply label these *current arrays* and *next arrays*.

Consider a forward traversal over the whole graph using the CSR layout. Assume all vertices are set in the frontier for reasons of simplicity. As such, the iteration visits all edges. The order in which edges are visited is given by the *destinations* array, where source vertices are implicitly assigned in increasing order. As such, the current arrays are traversed sequentially and have excellent spatial locality. The next arrays are accessed with a random access pattern and may have bad spatial and temporal locality, depending on the structure of the graph, especially for large-scale graphs.

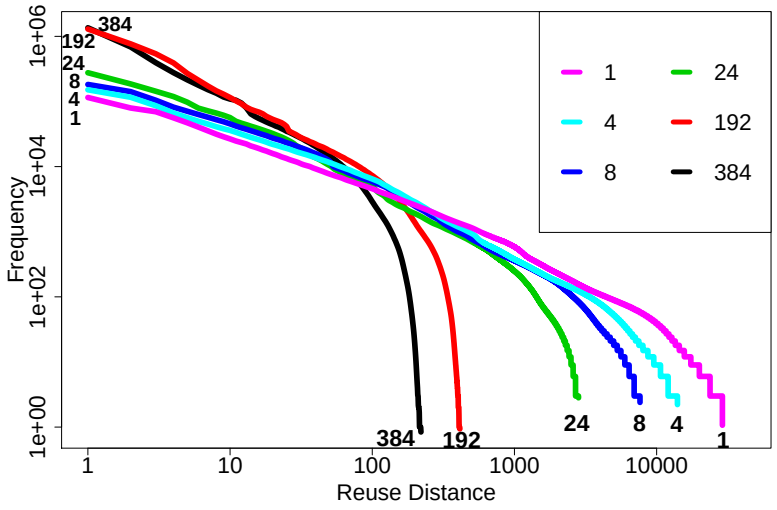


Figure 4.2: Reuse distance distribution of updates to the next frontier in PRDelta for the Twitter graph. The CSR layout is partitioned using partitioning-by-destination.

When partitioning the graph, in this example in two parts, it will independently traverse the partitions. Within each partition, we maintain sequential access to the current arrays and random access to the next arrays. The accesses to the next arrays are, however, confined to a subset of the vertices (those whose home partition is traversed). As such, the working set is smaller and temporal locality is improved. Looking across partitions, it traverses the current arrays twice (in general, once per partition). Each traversal is sequential but skips over some array elements. Depending on how many elements are skipped, spatial locality may be reduced.

Figure 4.2 shows the reuse distance distribution of accesses to the next frontier for the PRDelta algorithm applied to the Twitter graph (see Section 4.4 for details on graphs and algorithms). As the number of partitions is increased, the range of reuse distances contracts, and the frequency of reuse is increasing. This restriction to shorter reuse distances is an immediate consequence of restricting the range of

destination vertices. Moreover, the shorter reuse distances appear more frequently as the number of partitions is increased. The frequency of reuse is increasing from 10^4 to 10^6. Figure 4.2 also shows that the frequency becomes stable after 192 partitions. Hence, this chapter uses 384 as target number of partitions. This demonstrates the improved memory locality of graph partitioning.

The story is different for a backward traversal, which uses the CSC layout. Current arrays are accessed with a random access pattern, while next arrays are accessed sequentially. Due to partitioning by destination, however, edges are accessed in exactly the same order in the partitioned graph as in the whole graph. Partitioning-by-destination does not affect the memory locality of graph traversal. So, for CSC layout, GraphGrind uses the whole CSC graph layout with partitioned computation range.

Similar conclusions hold for partitioning-by-source: forward traversal visits edges in the same order as the whole graph while backward traversal observes improved temporal locality. Partitioning-by-source is not considered further in this dissertation as it incurs significant performance overhead. The overhead is the same in nature as that discussed below for partitioning-by-destination. However, as backward traversal is most beneficial when frontiers are rather sparsely populated, it almost always affects performance negatively.

4.2.2 Vertex Replication

Chapter 1 mentions that replication factor larger than one is a logical consequence of edge partitioning [33]. For instance, in Figure 4.1, the average replication factor is 7/6 (≈ 1.16) for the partitioned CSR layout.

Figure 1.1 measures more partitions up to 384 partitions, the replication factor grows slower than a linear function of the number of graph partitions. E.g., with 16 partitions, a vertex in the Twitter graph has outgoing edges in 5 out of 16 partitions on average. This is comparable to the vertex-cut algorithm [33]. This chapter needs a large number of partitions to improve temporal locality. However, section 2.4 presents that the replicate factor has adverse impact on memory usage and execution time.

4.2.3 Work Increase

Instruction count required to traverse the graph increases proportionally to the replication factor. Every vertex is visited as many times as it is replicated, i.e., $r(p)$ times on average. This replicates the actions of loading the vertex, checking its degree and whether it is active and iterating over (a subset of) its edges. As graph analytics typically require very little work per active edge, this control overhead has a noticeable impact already with 2 or 4 partitions.

The COO layout is again independent of vertex replication. The amount of work remains constant regardless of the number of partitions as each edge is visited once. The work is independent of how many vertices appear in a partition.

4.3 System Design

4.3.1 Graph Layout Options

The graph layout is tuned to the density of the frontiers. Contrary to prior work, which distinguishes between sparse and dense frontiers [43, 85, 108], this section

introduces a third case where frontiers are "*medium-dense*". The sparse frontier is a list of active vertex ID while a dense or medium-dense frontier is a bitmap.

4.3.1.1 Sparse Frontiers

When the frontier is sparse (typically less than 5% of vertices are active), little computation is done during graph traversal. A significant time spent is covered by overhead in control flow. In this case, there is little point in partitioning the graph [90]. As there is not much useful work performed, the opportunity to improve locality is low. Moreover, the state-of-the-art uses CSR in sparse frontiers because it allows to easily identify the active edges without having to traverse all (as opposed to CSC and COO) [85, 86]. As such, a copy of the *unpartitioned graph in CSR layout* is stored for the purpose of a traversal with sparse frontier.

4.3.1.2 Dense Frontiers

When the frontier is dense the majority of edges will be traversed. In this case, the COO layout is very efficient. The graph in a *large number of partitions in COO layout* is stored. The number of partitions is in principle bounded only by the size of the graph. It experimentally determines a good number of partitions.

4.3.1.3 Medium-Dense Frontiers

Here it introduces a new category of frontier density. A *medium-dense* frontier is dense enough to warrant representing the frontier as a bitmap, yet it is not dense enough to make a traversal over the COO layout fully efficient. Edge traversal on a medium-dense frontier is more efficient when using a CSR or CSC layout as it allows to skip over edges incident to inactive vertices. It will demonstrate

Algorithm 4 Edge-map decision procedure. $deg_{out}(v)$ is the out-degree of vertex v.

input : Graph $G = (V, E)$ in multiple formats, frontier F in bitmap format and operation *op*
side effect: Operation *op* applied to all outgoing edges of the active vertices in F
output : New frontier storing active vertices in bitmap

```
if |F| + Σ_{v∈F} deg_out(v) > |E|/2 then                    // dense frontier
    traversal of partitioned COO(G,F,op)
else if |F| + Σ_{v∈F} deg_out(v) > |E|/20 then        // medium-dense frontier
    backward traversal of unpartitioned CSC(G,F,op)
else                                                        // sparse frontier
    forward traversal of unpartitioned CSR(G,F,op)
end
```

that, when memory locality is addressed, algorithms with medium-dense frontiers perform best when using a backward traversal and a CSC layout. As partitioning by destination has no effect on the CSC layout, it is immaterial whether the CSC layout is partitioned or not. Finally, it chooses not partitioned.

4.3.2 Graph Traversal Decision Algorithm

The edge traversal procedure is summarised in Algorithm 4. It makes the following choices: (i) sparse frontiers traverse the whole graph; (ii) medium-dense frontiers use the CSC layout with a backward traversal; (iii) the most dense frontiers use the COO layout. It employs two experimentally defined thresholds to decide which traversal is most appropriate for each frontier. The 5% threshold for sparse frontiers is commonly used in the literature. It experimentally determined that a 50% threshold to differentiate medium-dense frontiers from dense frontiers works reliably across eight algorithms and eight graphs. The number of graph partitions in the CSC and COO layouts has an important impact on

performance. The best degree of partitioning differs between COO and CSC. As the COO layout takes storage independent of the number of partitions, we choose an aggressive partitioning degree.

This design trades off memory usage against execution speed. Where the state-of-the-art stores 2 copies of the graph (CSC and CSR) [8, 85, 108], It stores 3 copies. Note that the memory requirements are independent of the number of partitions as vertex replication does not increase memory consumption when partitioning the CSC and COO schemes by destination. As such, the memory requirement of our system is less than double the memory of Ligra.

4.3.3 Auxiliary Performance Benefits

Graph partitioning controls parallelism besides locality. When using *partitioning-by-destination* all the incoming edges of a vertex are located in the same partition and all partitions have **non-overlapping update sets**.

It ensures at least as many partitions as there are processing cores in the hardware. Each partition is processed by a single thread. Due to the non-overlapping nature of update sets of partitions, it can calculate updates **without using hardware atomic operations** such as compare-and-set. Atomic operations are very costly as graph analytics are already memory bound and the interconnection network between CPUs is highly loaded. It observed a speedup between 6.1% and 23.7% by removing atomic operations.

4.3.4 Implementation

These developed techniques are implemented in GraphGrind-v1 (NUMA) [90] of Chapter 3. The details of the baseline system are described elsewhere [90]. This chapter distinguishes the modifications presented below through the name "GraphGrind-v2 (Locality)". Performance will be compared against Ligra and Polymer to demonstrate that GraphGrind is a state-of-the-art graph analytics framework.

GraphGrind partitions graphs *by destination*. It distinguishes two distinct criteria to create balanced partitions, depending on the properties of the algorithm described in Chapter 3, *vertex-oriented* and *edge-oriented*. For *vertex-oriented* algorithms, GraphGrind loads balance the traversal such that each thread processes an equal number of distinct source vertices, as this correlates with the amount of work per traversal. For *edge-oriented* algorithms, it loads balance the traversal to balance the number of edges per thread. The COO layout is always partitioned such that each partition has the same number of edges.

Edge traversal using the dense operators are performed exclusively by CPU cores attached to the NUMA domain that stores the graph partition. Graph partitions are spread over all NUMA domains. As there is 4 NUMA domains on our experimental platform, we consider only multiples of 4 and allocate the same number of partitions on each NUMA domain. Frontiers represented as bitmaps and application-specific arrays storing attributes of vertices are allocated across the NUMA domains such that the attributes are stored on the NUMA domain that will update those values.

4.4 Experimental Evaluation

Evaluate the locality benefits of graph partitioning experimentally using 48 threads (we disregard hyperthreading due to its inconsistent impact on performance). GraphGrind has more partitions than the number of threads now. So, this experiment wants to show that the number of partitions is the key factor of graph analytics performance. Having a multiple of the number of threads helps with load imbalance. This has nothing to do with hyper threading. We use 48 threads to do simulation and disable hyperthreading. It has 256 GB of DRAM. It compiles all codes using the Clang compiler with Cilk support. It evaluates 8 graph analysis algorithms (Table 2.1), using 8 widely used graph data sets (Table 2.2). It exclusively presents results using 48 threads and presents averages over 20 executions. The analysis focusses primarily on the Twitter and Friendster graphs as these are the largest real-world graphs in our study. The other graphs respect the same conclusions.

4.4.1 Graph Layouts

This section investigates partitioning of graph layouts using the 8 graph algorithms and the Twitter graph (Figure 4.3). There are 4 configurations: the CSC and CSR layouts, the COO layout with atomic operations, and the COO layout without atomic operations. Atomics are unnecessary in the CSC layout due to grouping edges by their destination. They are unavoidable when using CSR due to partitioning by destination. Atomics can be avoided for COO when a partition is processed by one thread exclusively. In these experiments, this happens for 48 partitions or more as we evaluate using 48 threads.

The smallest partition number evaluated is 4 and corresponds to the left-most point on the curves. The CSC/CSR layout with 4 partitions is similar to Polymer and serves as a reference. With the CSC/CSR layout GraphGrind and Polymer quickly run out of memory. As such GraphGrind can evaluate at most 48 partitions for the Twitter graph on our machine.

Whenever we have 48 partitions or more, it can avoid the use of hardware atomics. These operations are demanding on the coherence protocol. As graph analytics are memory bound, hardware atomics have an important performance impact. Figure 4.3 shows that at 48 partitions avoiding atomics results in 6.1% to 23.7% speedup (comparing COO+a vs COO+na). The COO layout scales to a large number of partitions. All algorithms and graphs observe incremental performance benefits up to 384 partitions. Execution time starts to increase at 480 partitions due to increased scheduling overhead.

While prior work has labeled algorithms as *forward* (having faster traversal over CSR) or *backward* (having faster traversal over CSC), the results contradict the ruling classification reported in the literature and summarised in Table 2.1. In contrast, we observe that vertex- vs. edge-orientation explains the results. Vertex-oriented algorithms perform best when using the CSC layout, while edge-oriented algorithms perform best using the COO layout with a high number of partitions. Whether the CSR, CSC or COO layout performs better in other situations depends on a lot of factors including the density of frontiers, the cost of atomic operations, memory locality and, unavoidably, graph structure. Moreover, the CSC layout has no, or hardly any, improved memory locality due to partitioning for vertex-oriented algorithms. In contrast, there is a performance improvement by increasing the number of partitions for edge-oriented algorithms.

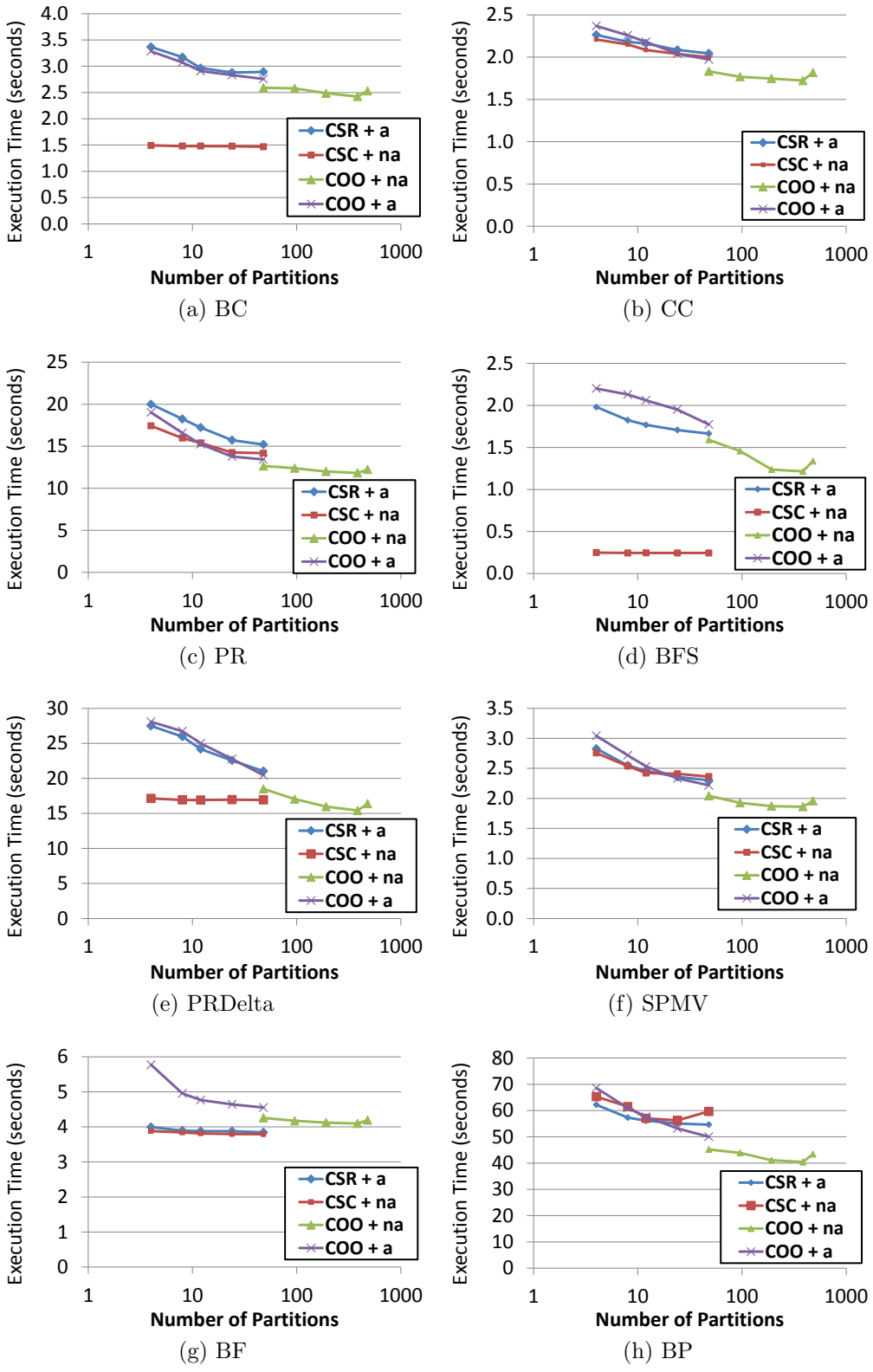
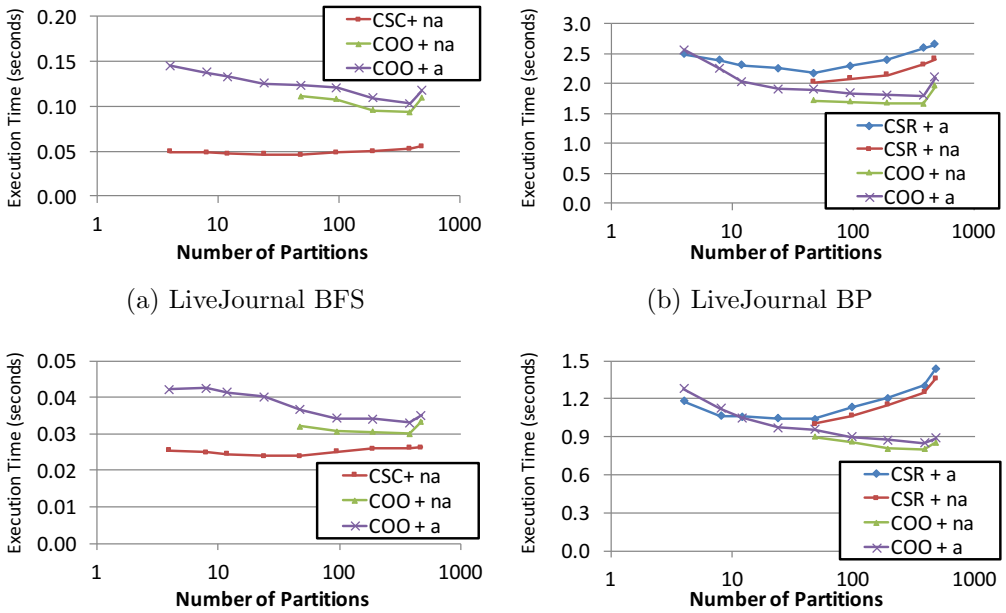


Figure 4.3: Execution time as a function of the number of partitions and graph layout for Twitter. "+a" is with atomics, "+na" is without atomics. Atomics can be disabled only when each partition can be processed sequentially by one thread.



(a) LiveJournal BFS

(b) LiveJournal BP

(c) Yahoo_mem BFS

(d) Yahoo_mem BP

Figure 4.4: Performance varying number of partitions for LiveJournal and Yahoo_mem. Backward traversal algorithms do not require atomics as the destination vertex is updated by a single thread. "+a" is with atomics, "+na" is without atomics.

This is not due to enhanced locality. Instead, it follows from improved load balancing. Parallelising traversal over a non-partitioned CSC or CSR implies that threads receive an equal number of vertices to traverse, which may correspond to a highly imbalanced number of edges per thread. Graph partitions, however, are designed to balance the edges, which enhances parallel efficiency.

The nature of vertex- vs. edge-orientation correlates strongly with the density of the frontier: in a vertex-oriented algorithm, updates of edges are by necessity propagated in a small fraction of the edges in order to achieve a time complexity proportional to the number of vertices. By not propagating most updates, the number of vertices activated each round will be small. As such, we can expect

the CSC layout to out-perform COO on some iterations, while COO is faster during iterations with more dense frontiers. This is the case for PRDelta, an edge-oriented algorithm, where 8 frontiers are dense, 3 are medium-dense and 22 are sparse.

4.4.2 Emulating Unrestricted Memory Capacity

The results indicate that the benefits of the CSR layout are restricted by memory consumption. In order to understand what would be feasible with more main memory, it takes a closer look at two small graphs: LiveJournal and Yahoo_mem (Figure 4.4). They can scale up the number of graph partitions in CSR layout. However, edge-oriented algorithms quickly see diminishing returns and a slowdown. This is a consequence of vertex replication which increases the amount of work. Vertex-oriented algorithms (e.g., BFS) do not observe significant performance variation when increasing the number of partitions. Other algorithms not shown here confirm these conclusions.

Ligra exploits parallelism between vertices, but for CSR case, Ligra needs atomics to avoid data race, e.g., different source vertices try to update computation values to the same destination vertex simultaneously. This leads to a high hardware overhead for larger graph. Figure 4.3 shows that the performance of more partitions without atomics is better. When there are more partitions than threads, and this dissertation uses partitioning by destination, we could ensure each memory location is updated by one thread only. In other words, this chapter changes the parallelism unit from vertex to partition. GraphGrind allows all the partitions to do computation in parallel, but the vertices in each partition will

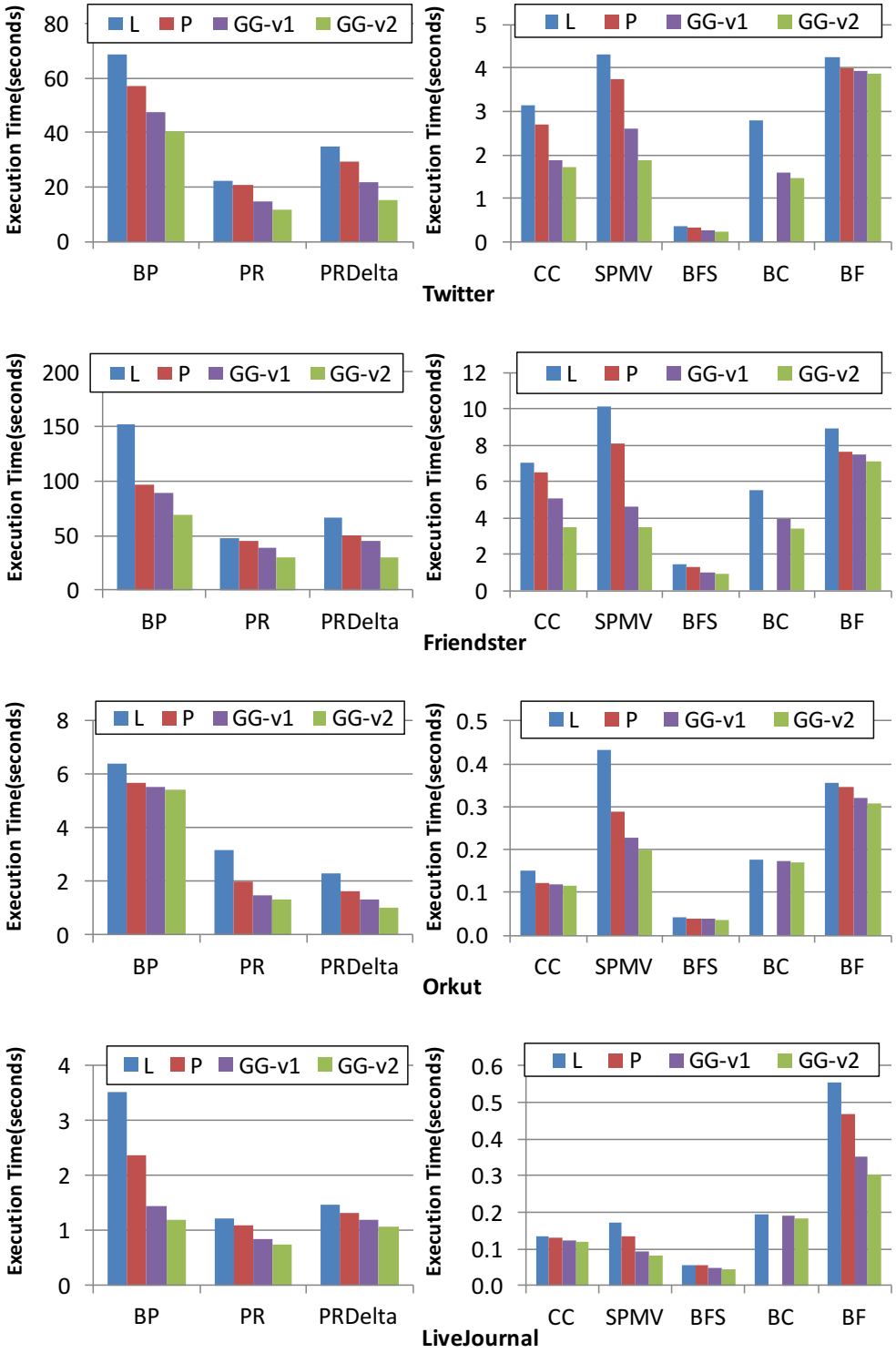


Figure 4.5: Comparison of our graph traversal algorithm against Ligra (L), Polymer (P) and GraphGrind-v1 (GG-v1). Polymer and GraphGrind-v1 use 4 partitions to match the number of NUMA nodes in our machine, GraphGrind-v1 only uses CSC/CSR format. GraphGrind-v2 (GG-v2) uses 384 partitions for the CSC computation chunk size and the COO layout.

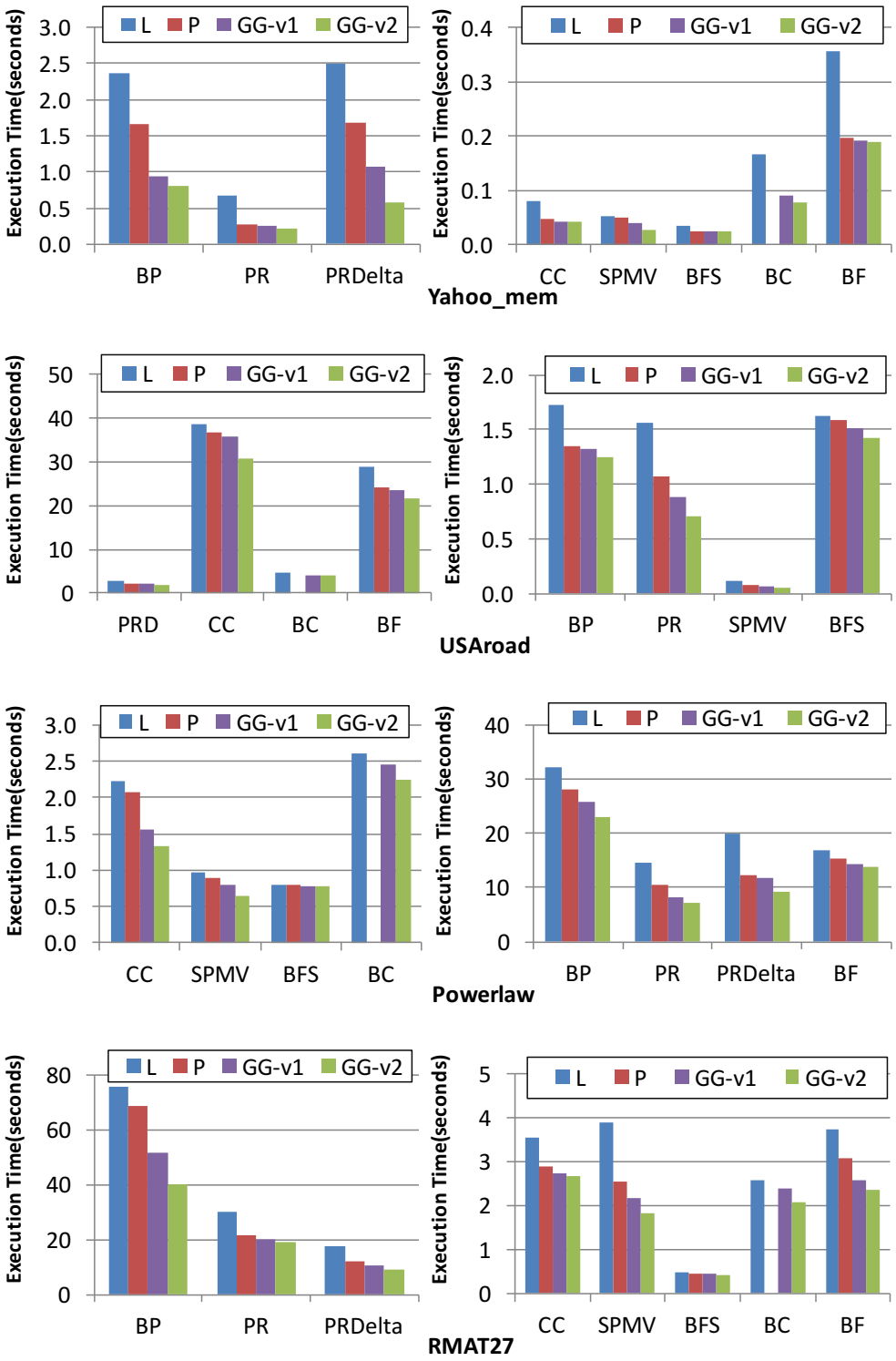


Figure 4.6: Comparison of our graph traversal algorithm against Ligra (L), Polymer (P) and GraphGrind-v1 (GG-v1). Polymer and GraphGrind-v1 use 4 partitions to match the number of NUMA nodes in our machine, GraphGrind-v1 only uses CSC/CSR format. GraphGrind-v2 (GG-v2) uses 384 partitions for the CSC computation chunk size and the COO layout.

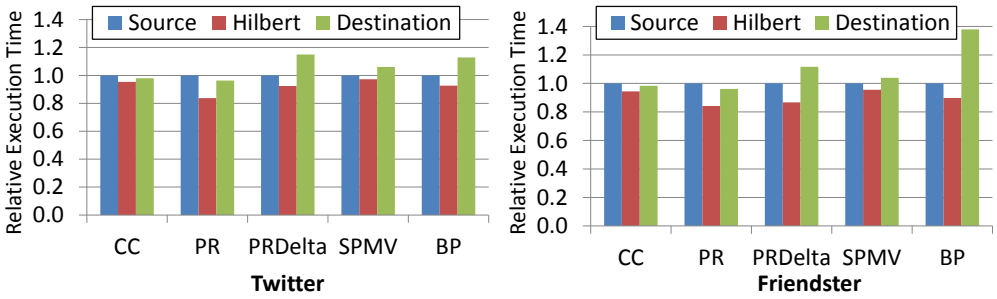


Figure 4.7: Performance impact of sort order of edges. Experiments performed for 384 partitions with 48 threads. Execution times are normalised to sorting edges by source (CSR order).

be traversed and updated sequentially. In this chapter, the number of partitions is up to 384 and the number of parallelism threads is 48. In all cases, avoiding atomics, which is possible with 48 partitions or more, reduces execution time. In fact, having at least as many partitions as threads is enough to avoid atomics. Here, the number of partitions is multiple of the number of threads in order to optimise load balance. Note that in BFS there is no need to use atomics in the CSC case as it uses a backward edge traversal.

4.4.3 Sorting Edge Lists

The COO layout may store edges in various orders. Up to now, One is in the same order as the CSR representation, i.e., they are sorted by the source vertex. Alternatively, edges may be sorted by destination vertex, which corresponds to the same traversal order as CSC. Another option is sorting edges using a space-filling curve such as Hilbert order to improve memory locality [66, 71].

Comparing these three options for the Twitter and Friendster graphs shows that Hilbert sorting has consistently lowest execution time (Figure 4.7). It is up to 16.2% faster. While graph partitioning makes a large improvement to memory

locality, traversing edges in Hilbert order further improves it.

Interestingly, the results reflect the preferred traversal order, showing that the CC and PR algorithms, which are edge-oriented backward algorithms, see better performance when sorting edges by destination (CSC order) compared to sorting by source. For the other algorithms, preferring forward traversal, the CSR order performs better. The performance benefit of backward traversal is explained in part by improved memory locality. Prior work has identified that backward traversal is sometimes faster than forward traversal, but did not provide a thorough explanation [8].

4.4.4 Memory Locality

Figure 4.8 shows last-level cache misses per kilo instructions (MPKI) for a varying number of partitions. MPKI values are high as graph analytics is a memory-intensive workload. However, graph partitioning significantly reduces the MPKI values. For instance, for PR on the Friendster graph, MPKI is reduced by half: from 29.0 at 4 partitions to 15.1 at 384 partitions. This demonstrates that shorter reuse distances (Figure 4.2) translate to fewer main memory accesses, and higher performance. For BFS, a vertex-oriented algorithm, graph partitioning does not reduce cache misses.

4.4.5 Comparison to State-of-the-Art

Finally GraphGrind-v2 compares against the state-of-the-art: Ligra [85], Polymer [108] and GraphGrind-v1 [90] (Figure 4.6). Results for BC on Polymer are missing as Polymer does not provide an implementation for this algorithm.

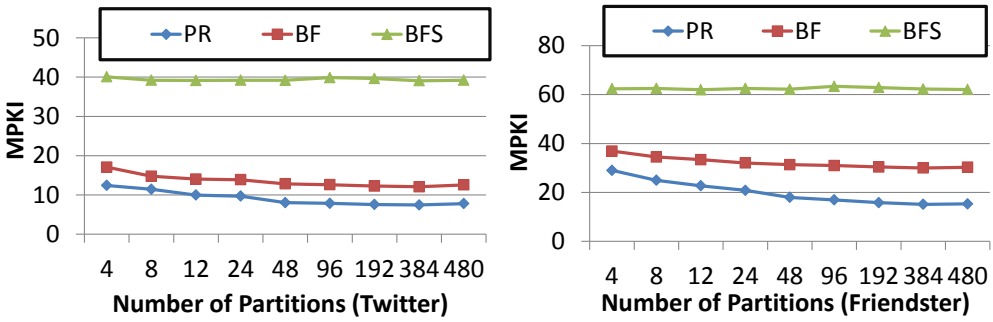


Figure 4.8: Misses per kilo instructions (MPKI) of Hilbert-sorted COO.

GraphGrind-v2 out-performs by a significant margin Ligra, Polymer and Graph -Grind-v1. For vertex-oriented algorithms, this is due to (i) using a non-partitioned CSR layout for traversals with sparse frontiers and (ii) enhanced memory locality by using 384 partitions in the CSC computation chunk size. Speedup for the vertex-oriented algorithms ranges from 2.35% to 37.31% over Ligra, from 0.28% to 19.9% over Polymer and from 0.6% to 15.1% over GraphGrind-v1.

Edge-oriented algorithms benefit from the enhanced memory locality due to graph partitioning and, to a lesser extent, sorting edges. Using a non-partitioned CSR layout for the sparse iterations is again beneficial. The speedup for PRDelta using our scheme is 138% for Twitter and 129% for Friendster. BP is sped up by 47.7% for LiveJournal and 108% for Yahoo_mem. The speedup ranges up to 4.34× over Ligra, up to 2.93× over Polymer and up to 45% over GraphGrind-v1, in both cases for PRDelta on Yahoo_mem.

The developed technique achieves high speedup on the USAroad graph, a road network graph that is hard to process for graph analytics frameworks. We achieve 23.9% speedup over Polymer for PRDelta and 22.6% for SPMV.

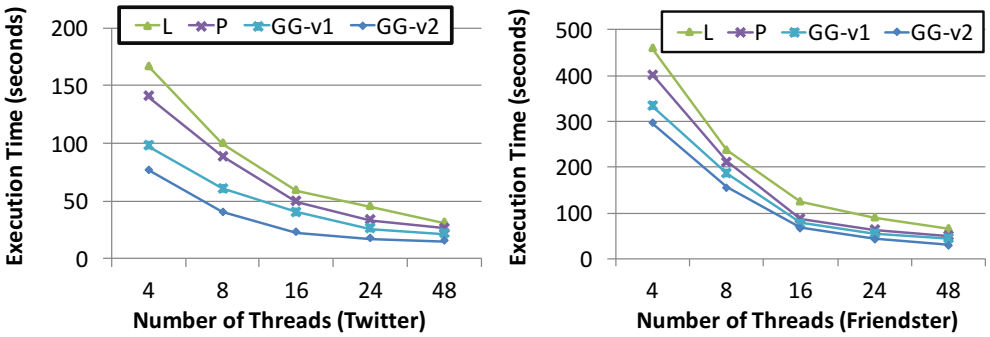


Figure 4.9: Parallel scalability compared to Ligra (L) Polymer (P) and GraphGrind-v1 (GG-v1) for PRDelta.

4.4.6 Parallel Scalability

Finally, Figure 4.9 demonstrates the scalability of our graph traversal algorithm for PRDelta. It shows execution time starting at 4 threads, as fewer threads do not work well with Polymer and our system as they require at least 1 thread per NUMA node. Additional threads are spread uniformly across NUMA nodes. Increasing parallelism reduces execution time commensurately. For instance, on the Friendster graph, Polymer speeds up 6x from 4 to 48 threads, while our system speeds up 10x.

4.4.7 Selecting The Degree of Partitioning

GraphGrind-v2 has a hidden parameter that determines how many partitions are employed for the COO layout. This parameter may be exposed to programmers and users. However, it would be convenient to determine them heuristically. Evaluation results show that graph partitioning scales to about 384 partitions for all graphs and algorithms. Further investigation is required to understand the mechanisms that determine performance as a function of partitioning degree.

4.5 Summary

This chapter differs from prior work by *improving temporal locality* through graph partitioning, which controls the order of iteration over edges. Each partition will be traversed per thread in parallel, and each vertex is updated sequentially per partition. It treats the partitioned graph as parallelism unit instead of the vertex, which avoids the use of costly hardware atomics. The partitioning algorithm is designed such that each vertex is updated by at most one thread, obviating the need for synchronization and the use of costly hardware atomics.

In order to scale to a large number of partitions, we use a combination of COO, CSR and CSC graph layouts. Prior work distinguishes between *sparse* and *dense* frontiers for optimization. Here identifies a third frontier type, *medium-dense* frontiers, which are highly populated but nonetheless benefit from an indexed graph representation such as CSR or CSC. This chapter proposes a fully automated selection of traversal direction based on the density of frontier iteratively, remove the requirement that programmers decide whether an algorithm runs faster when traversing the graph in a backward or in a forward direction. It is observed that this distinction is not accurate and show that the traversal order may be selected based on frontier density. Finally, GraphGrind achieves an average speedup of 1.79x over Ligra, 1.42x over Polymer and 1.22 over GraphGrind-v1 in Chapter 3.

An open question, which prior work has not answered in a conclusive manner [8, 85], is why a backward or forward traversal order is faster. We have demonstrated that frontier density and memory access order are important factors. A complete understanding of this subject may result in additional speedup

of graph analytics.

This chapter finds that the major distinction between dense, medium-dense and sparse frontier. The preferred traversal direction has become relevant for the size of frontier size and graph layouts.

Chapter 5

Using Vertex Reordering to Achieve Load Balance

Chapter 3 and Chapter 4 use a common goal, namely balancing the number of edges per partition. Other state-of-the-art schemes balance the number of edges in conjunction with other constraints such as minimizing the edge or vertex cut. This heuristic to balance the number of edges across partitions [33] is widely accepted [11, 54, 57], however, we have observed that this does not uniquely determine execution time. Moreover, this type of partitioning is computationally expensive. Moreover, while it is anticipated that the number of edges accurately predicts the computation load, such partitioning nonetheless results in computational load imbalance. This chapter presents an investigation on vertex ordering and proposes a new ordering algorithm to balance the number of edges per partitions as well as the number of unique destinations of those edges.

5.1 Introduction

In order to maximise processing speed, partitions should be chosen such that each partition takes the same amount of processing time. Moreover, the partitions should be largely independent; this minimises the volume of data communication. It has been demonstrated that partitioning the edge set is more effective than partitioning the vertex set [33], leading to the commonly used heuristics to strive for balanced edge partitions and minimising vertex replication (also known as ghost vertices) [11, 33, 57]. These constraints are typically mutually incompatible for scale-free graphs, resulting in a compromise between edge balance and vertex replication [11].

The quality of graph partitioning has a significant impact on graph processing speed. For instance, for an approximating PageRankDelta computation, one may observe that, during a first phase of the algorithm, about half of low-degree vertices converge before any high-degree vertex converges. A partition that consists of mostly high-degree vertices will thus take longer to process than a partition with only low-degree vertices, resulting in load imbalance. Note that it is likely to encounter partitions with mostly low-degree vertices in graphs with a power-law degree distribution as these graphs have many more low-degree vertices than high-degree vertices.

This chapter identifies that the time for processing a graph partition depends on both the number of edges and the number of unique destinations in that partition. This presents a new heuristic to partition graphs through ***joint destination vertex- and edge-balanced partitioning,*** which we call ***VEBO.***

A key motivation for considering joint vertex and edge balancing is provided by

the classification of *edge-oriented* algorithms and *vertex-oriented* algorithms [90] in Chapter 3. The properties of algorithms strongly affect partitioning: a vertex-balanced partition can result in almost a 40% speedup compared to edge-balanced partitioning for vertex-oriented algorithms [90]. While Chapter 3 and Chapter 4 select the partitioning heuristic depending on the algorithm, VEBO seamlessly resolves this important distinction between algorithm types.

Further motivation may be found by considering the amount of data that is updated during the processing of a graph partition, which is proportional to the number of unique destinations. As store operations in the memory system are often more expensive than load operations [108], the computation is more balanced when each partition has an equally large store set, even if this comes at the expense of a difference in the size of the read set.

This chapter also identifies a need to adapt graph partitioning to the characteristics of the graph processing system. Each system has its unique set of design choices, which determine its key performance bottlenecks. For instance, Ligra [85] uses dynamic scheduling to manage parallelism, but does itself not improve memory locality. In contrast, Polymer [108] and GraphGrind [90] carefully layout data in a NUMA-aware manner and use static scheduling in order to bind computation to the appropriate NUMA domains. Static scheduling makes parallel loops sensitive to load balance as the execution time of the loop is determined by the last-completing thread. It may thus be expected that graph partitioning serves different purposes: for Ligra, memory locality should be improved, while for Polymer and GraphGrind, load balance is more important.

This chapter proposes a graph partitioning algorithm that calculates an optimally load-balanced partition for power-law graphs with time complexity $\mathcal{O}(n \log P +$

$n \log n$), where n is the number of vertices in the graph and P is the number of partitions. Extensive experimental evaluation using three shared memory graph processing systems demonstrates a near-perfect computational load balance across a variety of graph data sets and algorithms. Contrary to heuristics such as Reverse Cuthill-McKee (RCM) [30] and Gorder [98] that aim to optimise memory locality, we obtain a consistent performance improvement when processing six scale-free graphs.

In summary, this chapter makes the following contributions:

1. It demonstrates the need to balance the number of unique destinations along with the number of edges in order to achieve computational load balance in parallel graph processing

2. A simple graph partitioning algorithm that optimally balances both edges and unique destinations using time proportional to $\mathcal{O}(n \log P + n \log n)$.

3. Addressing vertex-oriented and edge-oriented algorithms using a single graph partitioning heuristic

4. Extensive experimental evaluation using three shared memory graph processing systems (Ligra, Polymer and GraphGrind) and a comparison against edge balancing heuristics

The remainder of this chapter is organised as follows: Section 5.2 motivates the load balancing heuristic. Section 5.3 presents the VEBO algorithm and proves its optimality. Section 5.4 presents our experimental evaluation methodology. The experimental evaluation of VEBO is presented in Section 5.5.

5.2 Motivation

Section 2.4 shows that edge-balanced heuristic is locality-preserving in the sense that each partition consists of a chunk of consecutively numbered vertices. As such, the data for consecutive vertices are accessed by the same partition, which improves spatial locality. This algorithm is used in disk-based [54] and NUMA-aware graph processing [89, 90, 108].

Figure 5.1 shows the processing time of Chapter 4 for each of 384 partitions when executing one iteration of the PageRank algorithm. The graph is represented using the COO and edges are sorted in the access order of a Hilbert space filling curve in order to improve memory locality [71, 89]. Each partition is processed sequentially by one thread. This chapter studies the Twitter and Friendster graphs. Details on the graphs and experimental setup are provided in Section 3.4.

The top two plots in Figure 5.1 show that Algorithm 2 (Partitioning by destination) achieves a good edge balance. There is some variation on the number of edges in each partition, which is due to high-degree vertices. When high-degree vertices appear around the boundary between two partitions, there is no good decision as placing a high-degree vertex will either overload the first partition, or leave it underloaded. While partitions are edge-balanced well, the execution time per partition various over a factor of 6.9x and 2x for the Twitter and Friendster graphs, respectively. The VEBO heuristic, which we will present in the next Section, reduces the delay of the slowest partition over the fasted to a factor of 1.6x (Twitter) and 1.4x (Friendster).

The plots moreover show that the processing time of a partition is correlated to

the number of destination vertices (middle row), and of source vertices (bottom). Generally speaking, a partition with few destination vertices (and thus holding vertices with a high in-degree) is processed faster than a partition holding many low-degree vertices.

Note that VEBO results in a higher total number of source vertices. This occurs because Algorithm 2 (Partitioning by destination)partitions the set of destinations [108]. By consequence, sources are replicated in multiple partitions [33]. Figure 5.1f shows that VEBO increases vertex replication for Friendster.

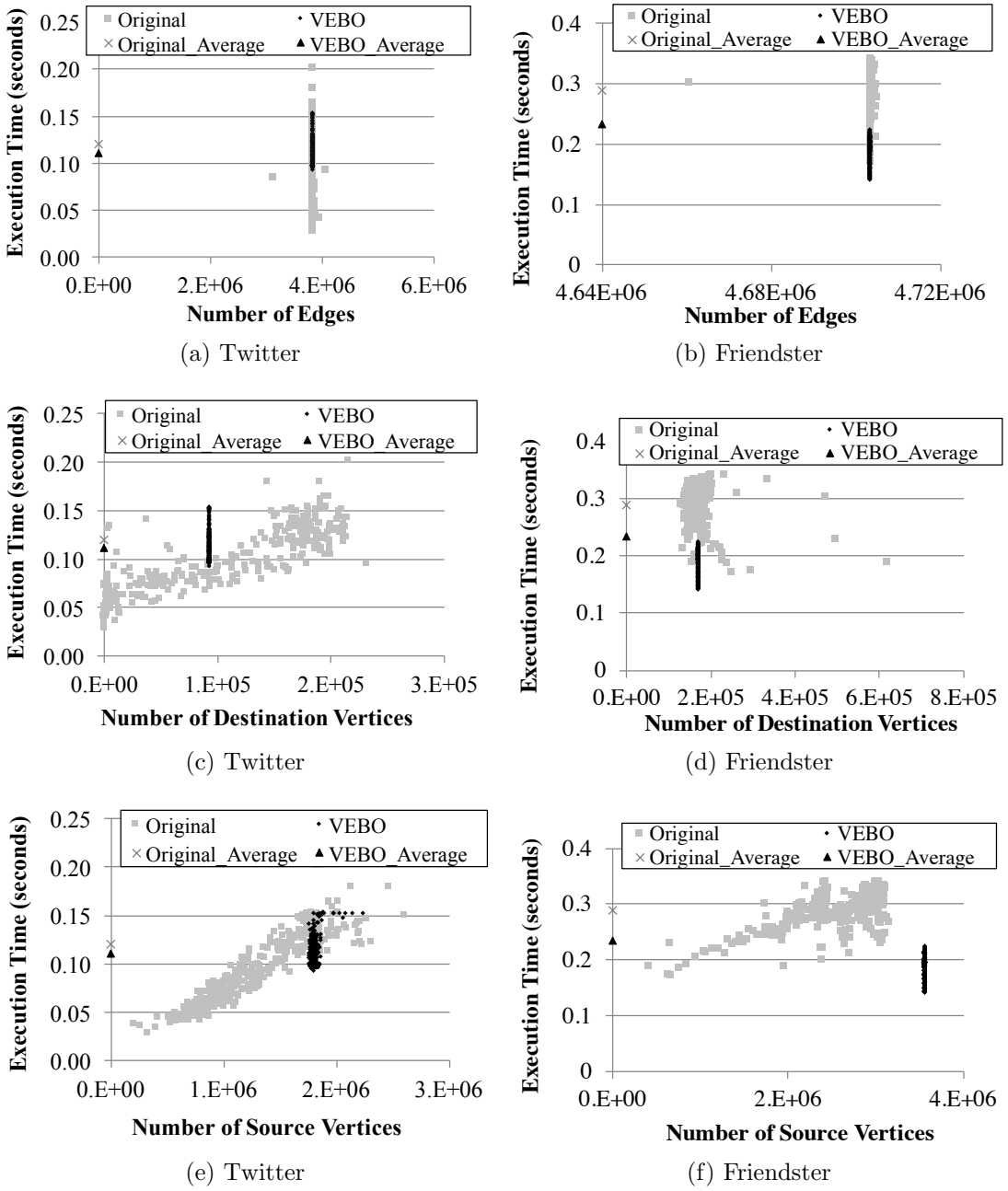
While both the number of unique source and destination vertices in a partition affects processing time, we choose to balance the number of destinations. This is most relevant from a practical point of view as partitioning of the destinations is used by a number of graph processing systems in order to create parallelism and avoid data races [54,90,108]. Alternatively, some systems partition the source vertices and replicate destinations [80]. For these systems, the analogous balancing criteria would focus on the source vertices.

5.3 The VEBO Algorithm

5.3.1 Problem Statement

Assume a graph $G = (V, E)$ with power-law in-degree distribution. Let N be one more than the highest in-degree in the graph. Let n be the number of vertices and let m be the number of vertices with non-zero in-degree.

There are a lot of suitable power-law distribution models, here, we only discuss one case. Model the in-degree distribution using a Zipf distribution where $s \geq 0$ is



(a) Twitter (b) Friendster

(c) Twitter (d) Friendster

(e) Twitter (f) Friendster

Figure 5.1: Processing time of a partition as a function of the number of edges, destinations, and source vertices in the partition. Each data point corresponds to one of 384 partitions. Average time of Twitter in original is 0.119s and in VEBO is 0.111s. Average time of Friendster in original is 0.289s and in VEBO is 0.234s.

the exponent governing the skewness of the degree distribution, N is the number of ranks and p_k, $k = 1, \ldots, N$ is the probability that a vertex has degree $k - 1$:

$$p_k = \frac{k^{-s}}{H_{N,s}} \tag{5.1}$$

where $H_{N,s} = \sum_{i=1}^{N} i^{-s}$ is a Generalised Harmonic Number. As such, the most frequent in-degree in the graph is degree zero, and least frequent in-degree that can be found in the graph is $N-1$. We make no assumptions about the out-degree distribution.

VEBO partitions the vertex set in P parts such that $V = \cup_{i=0}^{P-1} V_i$ and $V_i \cap V_j = \{\}$ if $i \neq j$. The partitions of the vertex set induce a partitioning of the edge set $E = \cup_{i=0}^{P-1} E_i$ such that for each $(v, w) \in E$: $(v, w) \in E_i$ if $w \in V_i$. The graph partitions are $G_i = (V, E_i)$, where any vertex can appear as a source vertex (hence the vertex set is V) but the set of destinations is restricted to V_i. The VEBO optimisation criteria are:

- minimise $max_{i=0}^{P}|E_i| - min_{i=0}^{P}|E_i|$ (*edge balance*)
- minimise $max_{i=0}^{P}|V_i| - min_{i=0}^{P}|V_i|$ (*vertex balance*).

5.3.2 Algorithm Description

The core idea of VEBO is the well-known heuristic for multiprocessor job scheduling, where the highest load in any partition is minimised [35]: place a set of objects in order of decreasing size, for each object selecting the least-loaded partition. Graham [35] has shown that the load of the highest-loaded partition is no worse than $4/3 - 1/(3P)$ times the highest load of the optimal placement. We

Algorithm 5 The VEBO reordering algorithm

input : Graph $G = (V, E)$; number of partitions P
output : Reordered sequence numbers $S[v] \in 0, \ldots, |V| - 1$ for $v \in V$ and partition end points $u[p] \in 0, \ldots, |V| - 1$ for $p \in 0, \ldots, P - 1$

29 Let w[P] = { 0 }; `// tracking edge count in each partition`
30 Let u[P] = { 0 }; `// tracking vertex count in each partition`
31 Let a[V] = { 0 }; `// assigned partition for each vertex`
32 Consider the list $v_{(0)}, v_{(1)}, \ldots, v_{(n)}$ of vertices sorted by decreasing in-degree, i.e., $\{v_{(0)}, v_{(1)}, \ldots, v_{(n)}\} = V$ and $deg_{in}(v_{(i)}) \geq deg_{in}(v_{(j)})$ when $i > j$;
Let n = $|V|$; `// the number of vertices`
33 Let m = $|\{v \in V : deg_{in}(v) > 0\}|$; `// the number of vertices` ; `// with non-zero degree`
`// Phase 1. Assign vertices with non-zero degree`
34 **for** $t \leftarrow 0$ **to** $m - 1$ **do**
35 Let $v = v_{(t)}$;
Let $p = \arg\min_{i=0,\ldots,P-1} w[i]$;
Let $a[v] = p$; `// assign` v `to partition` p
36 Increase $w[p]$ by $deg_{in}(v)$; `// update edge count`
37 Increase $u[p]$ by 1; `// update vertex count`
`// Phase 2. Assign vertices with zero degree`
38 **for** $t \leftarrow m$ **to** $n - 1$ **do**
39 Let $v = v_{(t)}$;
Let $p = \arg\min_{i=0,\ldots,P-1} u[i]$;
Let $a[v] = p$; `// assign` v `to partition` p
40 Increase $u[p]$ by 1; `// update vertex count`
`// Phase 3. Calculate new sequence numbers`
41 Let $s[0] = 0$;
for $p \leftarrow 1$ **to** $P - 1$ **do**
42 Let $s[p] = s[p - 1] + u[p - 1]$;
43 **for** $t \leftarrow 0$ **to** $n - 1$ **do**
44 Let $v = v_{(t)}$;
Let $S[v] = s[a[v]]$; `// determine sequence number for` v
45 Increment $s[a[v]]$ by 1;

will demonstrate that a much stronger bound may be obtained when the size of the objects follows a power-law degree distribution. Under this condition, the number of objects grows exponentially as their size reduces. As such, the numerous small objects can smoothen out any load imbalance caused by the (relatively)

few large objects, which results in an optimal load balance.

The VEBO algorithm (Algorithm 5 "The VEBO reordering algorithm") consists of three phases: In the first phase, VEBO places vertices with non-zero degree using the multiprocessor scheduling heuristic. This achieves a near-equal edge count in each partition. We will show that the edge imbalance is limited to 1 edge when the size of the placed objects follows a power-law distribution. In the second phase, VEBO places the zero-degree vertices. It builds on the same principle of power-law degree distribution, which implies that there are many zero-degree vertices. These vertices do not affect the edge balance. If any vertex imbalance is inadvertently introduced during the first phase, the vertex imbalance can be corrected by placement of the zero-degree vertices. In the third phase, VEBO reorders the vertices, i.e., it calculates a permutation of the sequence numbers of vertices, which is subsequently used to create a new representation of the graph.

5.3.3 Analysis

To support analysis of the VEBO algorithm, Appendix B introduces some auxiliary definitions. VEBO algorithm has been applied to 8 graphs with a selection of synthetic and real-world graphs. (Table 2.2). Seven graphs are power-law graphs. The USARoad graph represents a road network and has nearly constant degree. For all 8 graphs, VEBO calculates an optimal vertex- and edge-balanced placement with up to 384 partitions, i.e., $\Delta(n) = 1$ and $\delta(n) = 1$. This also holds for the USARoad graph.

5.3.4 Time Complexity

Algorithm 5 (The VEBO reordering algorithm) consists of three consecutive loops iterating over the vertices. In the first two loops, all statements take a constant number of time steps, except for the arg min operation which takes $\mathcal{O}(\log P)$ steps when implemented using a min-heap. As such these three loops take time $\mathcal{O}(|V| \log P)$. Algorithm 5 (The VEBO reordering algorithm) furthermore sorts the vertices in order of decreasing in-degree, which can be achieved in $\mathcal{O}(|V| \log |V|)$ using a variety of sorting algorithms. The total time complexity of the VEBO algorithm is thus $\mathcal{O}(|V| \log P + |V| \log |V|)$.

Algorithms for edge-balanced partitioning in conjunction with minimisation of vertex replication are computationally more complex. E.g., Gorder [98] takes $\mathcal{O}(\sum_{v \in V}(deg_{out}(v))^2)$ steps where $deg_{out}(v)$ is the out-degree of vertex v. The algorithm presented by Li *et al* [57] has polynomial time complexity in $|V|$. The time complexity of RCM is $\mathcal{O}(N \log N|V|)$ where N is the highest vertex degree.[3] The difference in complexity is evident as these algorithms solve a more complex problem. It will be demonstrated in the experimental evaluation that the partitioning criteria must be tuned to the properties of the graph processing system. As such, addressing load balance as VEBO can result in much higher efficiency compared to addressing vertex replication or memory locality.

[3] `http://www.boost.org/doc/libs/1_66_0/libs/graph/doc/cuthill_mckee_ordering.html`

Table 5.1: Key properties of Ligra, Polymer and GraphGrind for the purposes of this work.

System	Parallelism	Scheduling	Memory Opt.
Ligra	Cilk	dynamic	none
Polymer	Pthreads	static	NUMA
GraphGrind	Cilk w/ NUMA extension	mixed static and dynamic	NUMA and temporal locality

5.4 Evaluation Methodology

This chapter experimentally evaluates VEBO and compares it to two state-of-the-art graph reordering algorithms: RCM [30] and Gorder [98]. The RCM algorithm aims to reduce the bandwidth of a sparse matrix and is known to work well for applications in numerical analysis [30]. Gorder aims to improve temporal locality in graph analytics.

This chapter still uses three shared memory graph processing systems: Ligra [85], Polymer [108] and GraphGrind [89, 90] in Table 5.1. All systems implement the direction reversal heuristic [8] and dynamically adjust the frontier data structures depending on the frontier size [43]. The frontier varies during the execution and affects the best way to traverse the graph.

Ligra and Polymer do not distinguish dense from medium-dense frontiers and use a programmer-supplied flag to select forward or backward traversal direction for these frontiers. Chapter 4 shows that GraphGrind automatically tunes the traversal direction to the frontier density (Table 2.1). It uses a flag, however, to identify whether an algorithm is vertex-oriented or edge-oriented. This affects the partitioning algorithm, where either a vertex-balanced partitioning or an edge-balanced partitioning is applied. VEBO makes this flag redundant as a VEBO partition is both vertex-balanced and edge-balanced.

The key differences between these systems is in the scheduling of parallel work and memory locality optimisation (Table 5.1). Ligra expresses parallelism using Cilk [29], which is a fully dynamically scheduled parallel language. Ligra contains no specific optimisations for memory locality. Polymer expresses parallelism using POSIX threads and uses static scheduling, i.e., the amount of work performed by each thread is dictated by the size of the partition that it executes. Polymer optimises NUMA-locality, i.e., it maximises NUMA-local memory accesses compared to remote NUMA accesses. Like Ligra, it does not optimise temporal locality. GraphGrind [89] uses a mixture of static and dynamic scheduling. Static scheduling is used to bind partitions to NUMA sockets, while dynamic scheduling is used internally in a socket to distribute work across threads.

This chapter evaluates the performance benefits of VEBO experimentally using the same experiment environment, graph algorithms and data sets as Chapter 4. It exclusively presents results using 48 threads and presents averages over 20 executions.

The graph reordering algorithms are evaluated by first constructing reordered versions of the graphs. These are subsequently passed into the graph processing systems. Ligra does not partition graphs. Polymer and GraphGrind apply Algorithm 2 (Partitioning by destination) for graph partitioning. If the graph has been pre-processed by VEBO, it calculates partitions that are identical to VEBO. It generates 4 partitions with VEBO for Polymer, as it uses one partition per NUMA node and we generate 384 partitions with VEBO for GraphGrind, which is the recommended number of partitions [89]. Gorder graphs are generated by Gorder[4] [98].

[4]https://github.com/datourat/Gorder

5.5 Experimental Evaluation

5.5.1 Performance Overview

Table 5.2 and Table 5.3 show the execution time achieved with the original graph, RCM, Gorder and VEBO using each of the Ligra, Polymer and GraphGrind processing systems. The fastest execution time among the graph reordering algorithms in bold face, while slowdowns over the original graph are shown using italics.

Considering first the results for Ligra, we observe that the best graph order varies between algorithms and between graphs. Sometimes, all "optimised" graph orders result in worse performance compared to using the original graph (e.g., BFS on the Twitter graph).

RCM performs most consistently for the BC, BFS and BF algorithms. This may be explained as RCM places vertices in the BFS traversal order, and these three algorithms traverse the vertices in BFS order. As such, RCM prepares the graph order specifically for these algorithms. RCM also performs well for PR and SPMV, which always have a dense frontier. However, BP also always has a dense frontier, while RCM performs badly for 5 of the 6 graphs. Due to the heuristic nature of RCM, it is hard to find consistency in the data.

Gorder performs best for algorithms that execute mostly with dense frontiers, i.e., PR, PRdelta, SPMV, BF and BP. Gorder optimises memory locality, however, it assumes that all vertices and edges are active, as it does not know how the frontier will evolve during the algorithm. As such, the performance benefits are large for algorithms with dense frontiers. For algorithms with sparse frontiers, such as CC, BC and BFS, Gorder often results in a slowdown.

On Ligra, VEBO works reasonably well. It is the fastest option in 12 cases, while Gorder is fastest in 22 cases. However, VEBO is slower than no reordering in 10 cases while Gorder causes a slowdown in 15 cases. VEBO usually performs well for the RMAT27 graph, which suggests RMAT27 has strong load imbalance in its original order.

In summary, on the Ligra graph processing system, each graph ordering sometimes performs very well, but also often causes slowdown.

The performance of the graph ordering algorithms is very consistent on Polymer and on GraphGrind. VEBO addresses the load balancing problem effectively and consistently improves performance over the original graph. It results in the best performance for all graphs and algorithms. Gorder and RCM are less effective than VEBO for Polymer and GraphGrind. Due to the use of static scheduling, it is more important to improve load balance than to optimise memory locality. GraphGrind moreover optimises memory locality when loading the graph. It does so very effectively as the performance of GraphGrind on the original graph is almost always better than Ligra for each of the graph orders. As there is little scope to improve memory locality by graph reordering, neither Gorder nor RCM perform well.

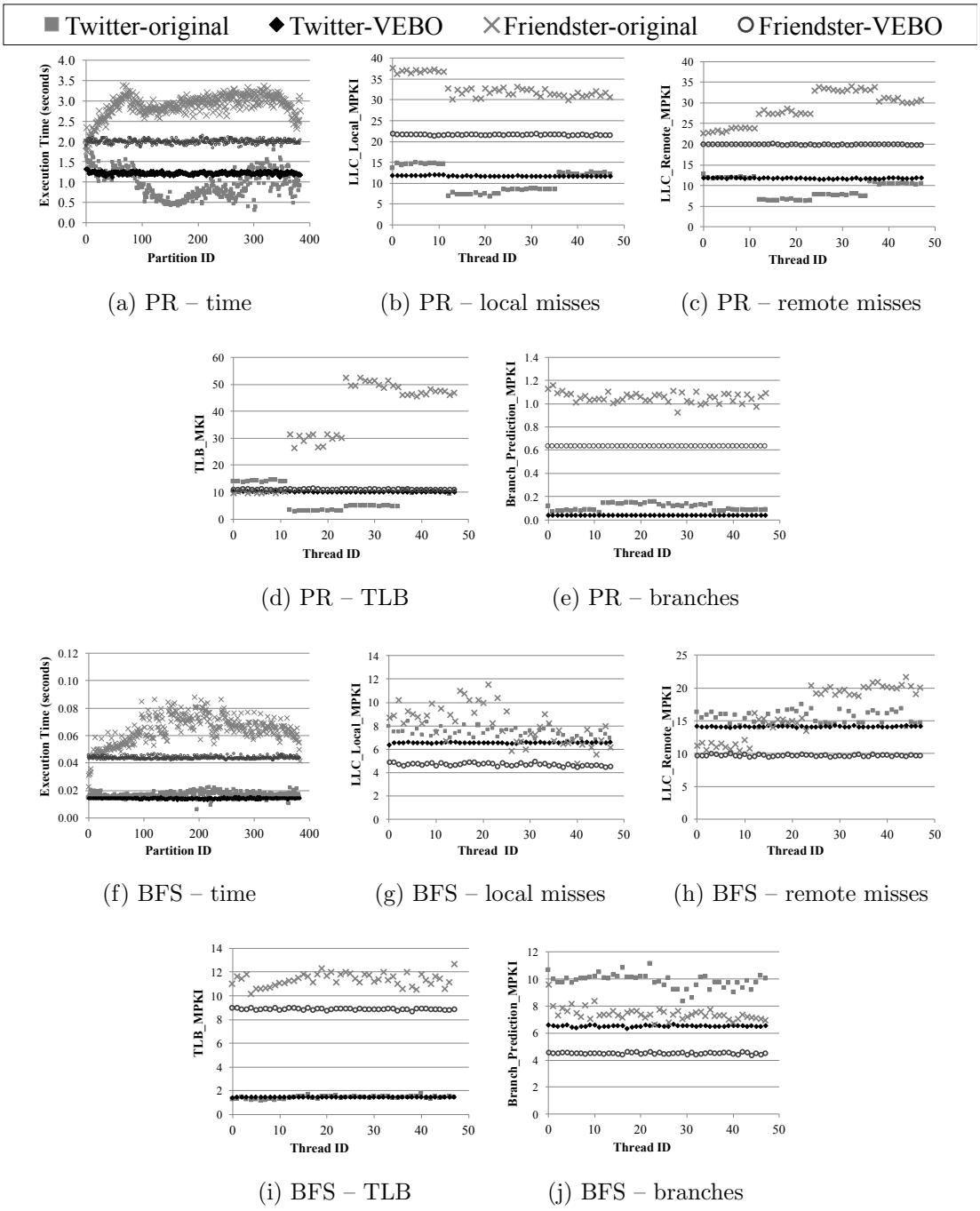
We conclude that when reordering graphs in order to increase processing speed, it is important to tune the graph order to address the main performance issues of the targeted graph processing system. For Ligra, the locality optimisation of Gorder performs better on the balance, while for Polymer and GraphGrind, load balancing is clearly more important.

Table 5.2: Runtime in seconds of Ligra, Polymer and GraphGrind using original graph, VEBO, Gorder and RCM. The fastest results of each framework are indicated in bold-face. Execution times that are slower than original graphs are indicated in italics.

		Ligra				**Polymer**				**GraphGrind**			
Algo.	**Graph**	**Orig.**	**VEBO**	**Gorder**	**RCM**	**Orig.**	**VEBO**	**Gorder**	**RCM**	**Orig.**	**VEBO**	**Gorder**	**RCM**
CC	Twitter	**3.132**	*4.062*	*3.217*	*4.242*	2.708	**2.353**	*2.972*	*2.930*	1.722	**1.329**	*2.261*	*2.250*
	Friendster	7.031	6.775	*13.517*	**6.636**	6.523	**5.296**	*10.035*	*6.775*	3.516	**3.391**	*8.216*	*3.530*
	RMAT27	3.544	**3.417**	*6.779*	*4.690*	2.880	**2.125**	2.543	2.622	2.656	**1.085**	2.198	2.511
	Orkut	0.151	**0.148**	*0.157*	*0.172*	0.123	**0.118**	*0.208*	*0.151*	0.116	**0.107**	*0.142*	*0.117*
	LiveJournal	0.133	*0.114*	**0.107**	*0.170*	0.133	**0.115**	*0.168*	0.129	0.120	**0.089**	*0.149*	0.116
	Yahoo_mem	0.080	0.044	0.045	**0.036**	0.049	**0.042**	*0.069*	*0.058*	0.042	**0.040**	*0.057*	*0.049*
BC	Twitter	2.798	**1.997**	*4.248*	2.492					1.478	**1.237**	*4.188*	*2.697*
	Friendster	5.499	4.411	5.038	**3.212**					3.428	**2.563**	*6.841*	2.947
	RMAT27	2.567	**2.330**	*5.991*	*3.544*					2.081	**1.387**	*5.943*	*3.207*
	Orkut	0.177	0.156	0.170	**0.141**					0.172	**0.147**	0.162	0.161
	LiveJournal	0.194	0.170	**0.128**	0.140					0.183	**0.155**	*0.210*	0.160
	Yahoo_mem	0.167	0.094	0.057	**0.052**					0.079	**0.075**	*0.089*	*0.088*
PR	Twitter	22.143	20.906	**20.142**	21.702	20.948	**16.903**	*25.934*	*22.271*	11.824	**10.726**	*16.219*	*11.979*
	Friendster	47.233	36.854	**29.532**	37.113	45.410	**28.776**	32.827	*46.573*	29.444	**20.053**	27.569	*29.981*
	RMAT27	29.965	**21.743**	*43.379*	27.012	21.645	**16.345**	19.278	19.958	19.250	**10.425**	15.602	15.395
	Orkut	3.184	2.548	**1.899**	2.569	2.003	**1.700**	1.888	1.893	1.337	**1.258**	1.275	1.288
	LiveJournal	1.219	0.931	**0.675**	*2.594*	1.080	**0.908**	1.010	*1.464*	0.751	**0.611**	0.634	*1.283*
	Yahoo_mem	0.684	0.283	**0.246**	0.285	0.274	**0.247**	*0.299*	*0.276*	0.226	**0.210**	0.242	0.215
BFS	Twitter	**0.347**	*0.445*	*0.581*	*0.699*	0.323	**0.303**	*0.336*	0.321	0.245	**0.217**	*0.249*	0.234
	Friendster	1.441	1.354	*1.620*	**0.808**	1.308	**1.103**	*2.214*	1.113	0.931	**0.554**	*1.890*	0.619
	RMAT27	**0.493**	*0.633*	*0.701*	*0.827*	0.456	**0.433**	0.443	0.440	0.405	**0.265**	0.284	0.271
	Orkut	0.041	*0.044*	*0.052*	**0.038**	0.039	**0.038**	*0.050*	*0.047*	**0.037**	**0.037**	*0.047*	*0.040*
	LiveJournal	0.056	0.048	**0.047**	**0.047**	0.055	**0.053**	*0.065*	*0.059*	0.046	**0.045**	*0.057*	*0.049*
	Yahoo_mem	0.035	0.026	**0.025**	0.026	0.025	**0.024**	0.025	*0.026*	0.024	**0.023**	**0.023**	*0.025*

Table 5.3: Continue. Runtime in seconds of Ligra, Polymer and GraphGrind using original graph, VEBO, Gorder and RCM. The fastest results of each framework are indicated in bold-face. Execution times that are slower than original graphs are indicated in italics.

		Ligra				**Polymer**				**GraphGrind**			
Algo.	**Graph**	**Orig.**	**VEBO**	**Gorder**	**RCM**	**Orig.**	**VEBO**	**Gorder**	**RCM**	**Orig.**	**VEBO**	**Gorder**	**RCM**
PR-Delta	Twitter	35.110	*39.627*	**32.695**	*43.617*	29.151	**20.205**	*33.336*	27.670	15.102	**12.013**	*19.613*	*15.352*
	Friendster	65.886	44.755	**41.021**	64.210	50.331	**40.377**	*56.370*	*63.000*	30.108	**22.011**	*33.223*	*36.666*
	RMAT27	17.688	**16.526**	*20.962*	*20.243*	12.134	**9.032**	*14.366*	11.034	9.002	**5.598**	*10.312*	8.601
	Orkut	**2.301**	*2.364*	*3.252*	*5.074*	1.630	**1.335**	*1.806*	*1.888*	1.038	**1.035**	*1.095*	*1.101*
	LiveJournal	1.464	*1.842*	**1.084**	*4.747*	1.320	**1.001**	*1.503*	*1.555*	1.061	**0.785**	*1.172*	*1.100*
	Yahoo_mem	2.501	2.479	**1.607**	*2.590*	1.687	**1.322**	1.533	*2.932*	0.676	**0.650**	0.665	*2.080*
SPMV	Twitter	4.311	1.959	**1.602**	2.281	3.746	**2.336**	3.183	3.293	1.861	**0.722**	1.186	1.199
	Friendster	10.112	**3.412**	7.300	3.730	8.124	**4.664**	*9.934*	6.337	3.511	**1.206**	*5.893*	2.051
	RMAT27	3.883	**2.470**	3.188	2.657	2.538	**2.228**	2.331	2.443	1.814	**0.602**	1.364	1.506
	Orkut	0.431	0.382	**0.104**	0.116	0.288	**0.256**	0.277	0.263	0.199	**0.065**	0.089	0.069
	LiveJournal	0.171	**0.055**	0.102	0.119	0.134	**0.099**	0.112	0.122	0.082	**0.035**	0.064	0.071
	Yahoo_mem	0.053	0.025	0.046	**0.020**	0.049	**0.041**	0.042	0.045	0.029	**0.018**	0.019	0.021
BF	Twitter	4.255	**3.640**	*9.394*	*4.649*	3.990	**3.566**	*11.246*	*9.325*	3.877	**2.872**	*9.907*	*7.059*
	Friendster	8.884	8.687	**6.136**	6.161	7.653	**7.036**	7.366	7.312	7.105	**6.134**	6.264	6.255
	RMAT27	3.718	2.791	**2.196**	3.320	3.090	**2.676**	*4.672*	*4.331*	2.352	**1.959**	*3.320*	*3.247*
	Orkut	0.357	0.322	**0.279**	*0.386*	0.345	**0.302**	0.331	*0.396*	0.306	**0.203**	0.258	*0.377*
	LiveJournal	0.555	**0.280**	0.464	0.356	0.468	**0.403**	*0.534*	0.577	0.305	**0.260**	*0.365*	*0.376*
	Yahoo_mem	0.357	0.291	0.237	**0.166**	0.197	**0.187**	*0.312*	0.191	0.188	**0.154**	*0.276*	0.166
BP	Twitter	68.767	*88.439*	**64.242**	*105.223*	57.310	**46.187**	53.365	50.398	40.412	**22.434**	36.314	31.850
	Friendster	151.742	112.785	**99.292**	132.865	96.113	**73.220**	75.392	85.475	69.526	**48.071**	48.586	54.147
	RMAT27	75.336	72.552	**70.729**	*76.805*	68.324	**42.403**	58.035	50.440	40.092	**18.409**	31.763	29.702
	Orkut	6.372	**5.654**	*8.107*	*12.643*	5.652	**4.052**	4.522	4.199	5.392	**1.547**	2.176	1.732
	LiveJournal	3.522	*3.754*	**1.880**	*4.335*	2.376	**1.745**	1.886	2.114	1.190	**0.625**	0.750	*1.735*
	Yahoo_mem	2.372	1.628	**1.235**	1.659	1.667	**0.656**	0.702	0.896	0.802	**0.275**	0.302	0.494



(a) PR – time (b) PR – local misses (c) PR – remote misses

(d) PR – TLB (e) PR – branches

(f) BFS – time (g) BFS – local misses (h) BFS – remote misses

(i) BFS – TLB (j) BFS – branches

Figure 5.2: Execution time and micro-architectural statistics per partition or per thread for Twitter and Friendster, and for PR and BFS. Measured on GraphGrind using 384 partitions. Thread t executes partitions $8\,t$ to $8\,t + 7$. Architectural statistics expressed in misses per thousand instructions (MPKI).

5.5.2 Analysis of Load Balance

This section analyses the load balance of the partitions constructed by VEBO for two graphs, Twitter and Friendster, and two algorithms, PR and BFS. It performs this analysis using GraphGrind and focus on the iterations with dense and medium-dense frontiers. PR is an edge-oriented algorithm where every edge is active during every iteration. It has only dense frontiers. BFS is a vertex-oriented algorithm where every vertex is active exactly once. It has no dense frontiers, so it is indicative of iterations with medium-dense frontiers.

Figure 5.2a and 5.2f show the execution time for each of the 384 partitions. There is a large variation on the execution time for the original graph, e.g., from 0.290 s per iteration to 1.985 s for PR on the Twitter graph. For the VEBO reordered graph (darkest symbols), the worst-case difference between the fastest and slowest partition is 0.15 s or more than 10 times less than the original graph. For most graphs and algorithms, we observe that the average time to process a partition is reduced by VEBO, compared to the original graph. This contributes to the speedup observed by VEBO, besides achieving load balance. VEBO does not achieve a speedup per partition in all cases. For instance, when processing PR for Twitter, the average execution time per partition is 1.121 s for VEBO and 0.928 s for the original graph.

Figure 5.2 furthermore shows that VEBO balances execution at the micro-architectural level. Table 5.5 shows the summary statistics across the edgemap operation. It shows NUMA-local cache misses (cache misses resolved in the local NUMA node); NUMA-remote cache misses (cache misses for data stored in a remote node); TLB misses and conditional branch mispredictions. These statis-

tics are collected per thread and correspond to the consecutive execution of 8 subsequent graph partitions.

In all cases, VEBO balances miss rates for caches, TLBs and branch predictors. When processing PageRank for Friendster, local and remote cache misses as well as TLB misses are significantly reduced. This shows that VEBO can improve memory locality, even though this was not part of the optimisation criterion We observed that VEBO improves memory locality for the majority of the graphs. VEBO however, reduces memory locality on PageRank for Twitter as cache and TLB miss rates are close to the worst case for the original graph.

While locality of the Twitter graph is reduced for PR, it is improved for BFS. As such, a graph order that improves memory locality for a vertex-oriented algorithm may be different from a graph order that improves locality for an edge-oriented algorithm. This is an interesting observation that merits further investigation. If true, it would imply that graph reordering for locality needs to take into account not only the graph structure but also properties of the algorithm that will process it.

Finally, it shows that VEBO reduces the branch misprediction rate (Figures 5.2e and 5.2j). We attribute this in part to ordering vertices by decreasing degree. GraphGrind uses the CSR and CSC data structures when frontiers are sparse or medium. For CSR and CSC, the code to iterate over the graph contains a loop over the edges incident to a vertex. The loop iteration count is determined by the degree. In the VEBO ordered graph, subsequent vertices have the same degree which makes this branch highly predictable. In the original graph, subsequent vertices have highly varying degrees, which makes it hard to predict the loop termination accurately.

Table 5.4: Architectural events for PR and BF on Twitter and Friendster. Results expressed in misses per thousand instructions (MPKI).

Graph	App.	Ori./VEBO	Vertex Map			
			Local misses	Remote misses	TLB mispred.	Branch misses
Twitter	PR	original	4.483	4.113	0.015	0.186
		VEBO	6.939	1.561	0.012	0.148
	BF	original	2.491	2.013	0.027	7.512
		VEBO	3.608	0.513	0.015	5.551
Friendster	PR	original	8.323	3.325	0.010	0.161
		VEBO	9.283	2.263	0.009	0.137
	BF	original	5.981	1.509	0.024	12.705
		VEBO	6.614	0.839	0.014	9.936

There are some cliffs in the TLB and cache statics, the reason is that the measurements are based on the thread number (thread ID). Each thread has consecutive eight partitions (384 partitions/48 threads) in fact during parallelism. There is imbalance in the number of vertices or edges with edge or vertex balance partitioning. For instance, twitter graph has a huge imbalance in the number of vertices using edge-balance partitioning. So, there are some cliffs between threads in the original graph (baseline). This chapter removes it for better load balance.

5.5.3 Edgemap vs Vertexmap

Graph algorithms expressed for Ligra, Polymer and GraphGrind use two key operations: *edgemap* and *vertexmap*. Table 5.4 shows the performance characteristics of the PageRank (PR) and Bellman-Ford (BF) algorithms. This section uses Bellman-Ford as an example of a vertex-oriented algorithm as BFS does not utilise vertexmap. Contrary to BFS, BF spends the majority of the execution using medium-dense frontiers (5–50% frontier density). VEBO simultaneously balances edges and vertices. As such, it optimises both the execution of edgemap

Table 5.5: Architectural events for the PR and BFS algorithms when processing Twitter and Friendster. Results expressed in misses per thousand instructions (MPKI).

Graph	App.	Ori./VEBO	Edge Map			
			Local misses	Remote misses	TLB mispred.	Branch misses
Twitter	PR	original	11.102	9.345	8.346	0.109
		VEBO	12.078	12.944	13.566	0.051
	BFS	original	7.389	15.515	1.432	9.881
		VEBO	7.419	15.110	2.280	7.180
Friendster	PR	original	33.000	28.746	34.760	1.056
		VEBO	24.598	28.712	15.297	0.798
	BFS	original	8.090	16.426	11.362	7.459
		VEBO	5.328	11.804	10.929	4.697

and vertexmap. However, the optimisation effects are different.

GraphGrind spreads the iterations of the vertexmap loop equally across all threads [90]. Arrays accessed by vertexmap, however, are distributed over the NUMA nodes according to the graph partitions. This cause a high number of remote cache misses because Algorithm 2 (Partitioning by destination) induces imbalance in the number of vertices per partition. When ordering the graph with VEBO, all partitions have an equal number of vertices. As such, each thread mostly accesses NUMA-local data, which explains the reduction in remote misses.

5.5.4 Space Filling Curves

GraphGrind sorts edges in the COO representation, which is used during dense frontiers, using the Hilbert space filling curve. This gives a significant performance boost [89]. However Hilbert order is a heuristic, which has been studied mostly for dense matrix algebra [15, 21, 67]. The followings are founded through studying graph reordering that (i) there exist cases where Hilbert order degrades performance; (ii) the effectiveness of Hilbert sorting depends on the number of

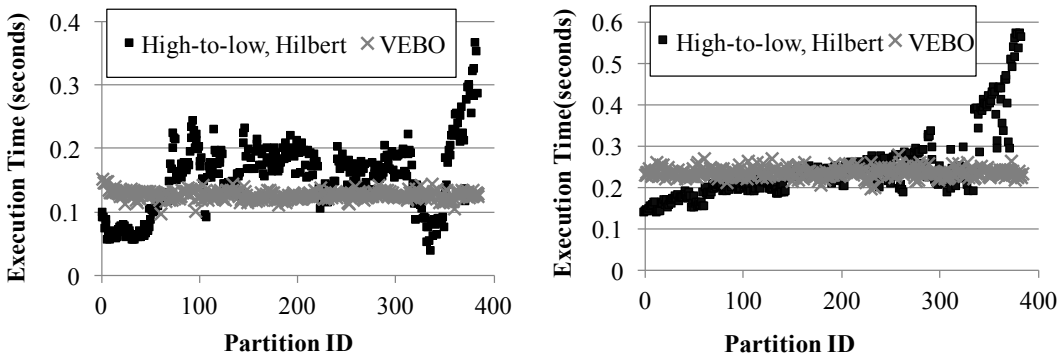
Table 5.6: Runtime in seconds of GraphGrind with VEBO when sorting edges in the COO format in the order of a Hilbert space filling curve (SFC) or in the CSR order. The fastest option is indicated in bold-face for each graph and algorithm.

Graph	COO	CC	PR	PRDelta	SPMV	BP
Twitter	SFC	1.329	10.726	12.013	0.722	22.434
	CSR	**1.115**	**8.592**	**9.717**	**0.719**	**21.150**
Friendster	SFC	3.391	20.053	22.011	1.206	48.071
	CSR	**3.070**	**14.602**	**16.738**	**1.018**	**46.242**
RMAT27	SFC	1.085	10.425	5.598	0.602	18.409
	CSR	**1.049**	**7.477**	**4.315**	**0.600**	**16.351**
Orkut	SFC	0.107	1.258	1.035	0.065	1.547
	CSR	**0.101**	**1.149**	**1.008**	**0.061**	**1.372**
LiveJournal	SFC	0.089	0.611	0.785	0.035	0.625
	CSR	**0.080**	**0.605**	**0.754**	**0.031**	**0.613**
Yahoo_mem	SFC	0.040	0.210	0.650	0.018	0.275
	CSR	**0.039**	**0.209**	**0.641**	**0.015**	**0.271**

non-zeroes. To the best of our knowledge, neither of these properties are adequately covered in the literature.

It first sorts all vertices from high in-degree to low and partition the resulting graph using Algorithm 2. (Partitioning by destination) We compare the performance of this high-to-low order against VEBO, both using 384 partitions, for PageRank (Figure 5.3). VEBO constructs partitions with a balanced mix of high- and low-degree vertices. The high-to-low order creates skewed partitions: The first partitions contain the vertices with the highest degrees. These are processed faster than a partition with mixed degrees (VEBO). The last partitions in the high-to-low order contain exclusively degree-one vertices and are processed up to three times slower than an average partition. Note that all partitions are edge-balanced. The performance is variable for the middle range of partitions due to the impact of the out-degrees of source vertices on performance.

This demonstrates that Hilbert order is more effective when the in-degree of vertices is high. High degrees imply more opportunity for reuse of data, which



(a) Twitter

(b) Friendster

Figure 5.3: Processing speed as a function of the in-degree of vertices. Showing first iteration of PR.

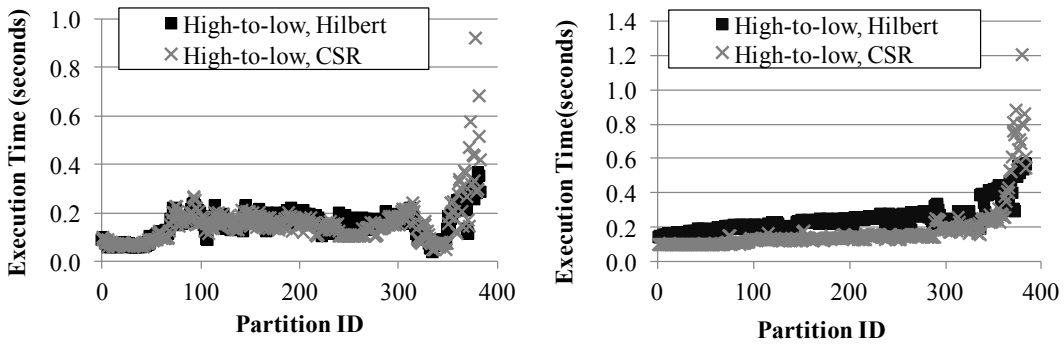
admit memory access order optimisation.

The second experiment aims to understand how effective Hilbert order is depending on the vertex degrees. This section again reorders the graph using the high-to-low order and compare Hilbert order to the traversal order of CSR, i.e., by increasing source vertex ID (Figure 5.4). Surprisingly, the CSR order admits faster processing for partitions 0–350 (approximately). Thus, for higher degrees, the CSR order is more efficient than Hilbert order. For very low degrees, Hilbert order is more efficient than CSR order. Note, however, that regardless of traversal order used, very sparse partitions are accessed far less efficiently than the average partition constructed by VEBO.

Putting these observations together, traversing edges in CSR order is more efficient than Hilbert order for a wide range of vertex in-degrees. Hilbert order is significantly faster only when there are many low-degree vertices in a partition. The original graphs have widely varying degree distributions across partitions and Hilbert order works on average better than CSR order for the original graphs [90].

Table 5.7: Runtime in seconds of Ligra, Polymer and GraphGrind using original graph, VEBO, Gorder and RCM. The table is constructed similarly to Table 5.2.

		Ligra				**Polymer**				**GraphGrind**			
Algo.	**Graph**	**Orig.**	**VEBO**	**Gorder**	**RCM**	**Orig.**	**VEBO**	**Gorder**	**RCM**	**Orig.**	**VEBO**	**Gorder**	**RCM**
CC	USAroad	38.669	11.961	**7.953**	*54.119*	36.877	12.037	**7.834**	*46.338*	30.754	9.317	**7.709**	*41.188*
	Powerlaw	**2.229**	*15.533*	*5.003*	*9.575*	**2.070**	*12.303*	*4.322*	*8.337*	**1.322**	*10.106*	*3.557*	*5.465*
BC	USAroad	**4.620**	*5.297*	*4.964*	*4.783*					**3.892**	*4.340*	*4.198*	*4.443*
	Powerlaw	**2.603**	*6.802*	*3.062*	*6.601*					**2.252**	*4.502*	*2.522*	*4.982*
PR	USAroad	**1.559**	*2.321*	**1.559**	*1.855*	**1.075**	*1.662*	*1.382*	*1.703*	**0.707**	*1.288*	*1.282*	*1.330*
	Powerlaw	**14.619**	*24.425*	*18.463*	*36.164*	**10.495**	*21.604*	*17.227*	*33.371*	**7.323**	*16.216*	*14.534*	*27.732*
BFS	USAroad	**1.621**	*1.653*	*1.937*	*1.819*	**1.588**	*1.591*	*1.819*	*1.766*	**1.424**	*1.581*	*1.728*	*1.586*
	Powerlaw	**0.794**	*3.212*	*1.282*	*3.242*	**0.804**	*1.974*	*2.037*	*2.412*	**0.775**	*1.432*	*1.799*	*1.912*
PR-Delta	USAroad	**2.886**	*4.666*	*2.975*	*2.982*	**2.241**	*2.997*	*2.584*	*2.733*	**1.809**	*2.905*	*1.828*	*2.044*
	Powerlaw	**19.925**	*25.393*	*24.236*	*74.178*	**12.331**	*23.120*	*21.375*	*53.348*	**9.197**	*20.637*	*18.107*	*37.741*
SPMV	USAroad	**0.120**	*0.203*	*0.143*	*0.135*	**0.079**	*0.094*	*0.132*	*0.115*	**0.053**	*0.083*	*0.110*	*0.103*
	Powerlaw	**0.967**	*2.341*	*1.707*	*3.510*	**0.893**	*1.890*	*1.107*	*4.279*	**0.644**	*1.699*	*0.932*	*1.794*
BF	USAroad	**28.848**	*37.503*	*34.646*	*29.175*	**24.067**	*35.233*	*31.334*	*28.336*	**21.510**	*31.110*	*25.976*	*26.334*
	Powerlaw	**16.866**	*22.719*	*19.070*	*30.725*	**15.226**	*45.327*	*27.322*	*39.672*	**13.844**	*44.032*	*22.057*	*31.690*
BP	USAroad	**1.730**	*4.059*	*1.785*	*1.843*	**1.343**	*1.368*	*1.391*	*1.543*	**1.245**	*1.441*	*1.258*	*1.334*
	Powerlaw	**32.119**	*43.596*	*36.684*	*120.040*	**28.114**	*41.626*	*34.667*	*93.667*	**23.082**	*37.037*	*32.147*	*59.116*



(a) Twitter

(b) Friendster

Figure 5.4: Processing speed of Hilbert order and CSR order as a function of the in-degree of vertices. Showing first iteration of PageRank.

As VEBO creates nearly the same degree distribution in each partition, it is expected that CSR order is more efficient than Hilbert order.

GraphGrind is modified accordingly to change the sort order in the COO representation to CSR order. This change consistently speeds up the five algorithms that contain dense frontiers (Table 5.6). Speed-ups range up to 37% (PR, Friendster) and 39% (PR, RMAT27).

5.5.5 Hard Graphs

Two graphs stand out as hard graphs to improve performance by reordering: the Powerlaw graph and the USARoad graph. Their performance is shown in Table 5.7, which has the same structure as Table 5.2 and Table 5.3.

This chapter has analysed the reason behind the slowdowns for the Powerlaw graph and found that memory locality was significantly affected by reordering, for all of Gorder, RCM and VEBO. Note that VEBO did achieve optimal load balance. Gorder cannot improve memory locality for this graph. It needs to

Table 5.8: Break-down of iterations of Connected Components on the USARoad graph. Data collected with GraphGrind using 384 threads. Showing number of iterations (Iter.) and execution time in seconds (Exec.).

Graph Order	Dense		Medium		Sparse	
	Iter.	Exec.	Iter.	Exec.	Iter.	Exec.
Original	4	0.241	240	24.868	5689	3.122
Gorder	5	0.264	63	3.244	6673	4.166
VEBO	6	0.284	78	4.522	6744	4.293

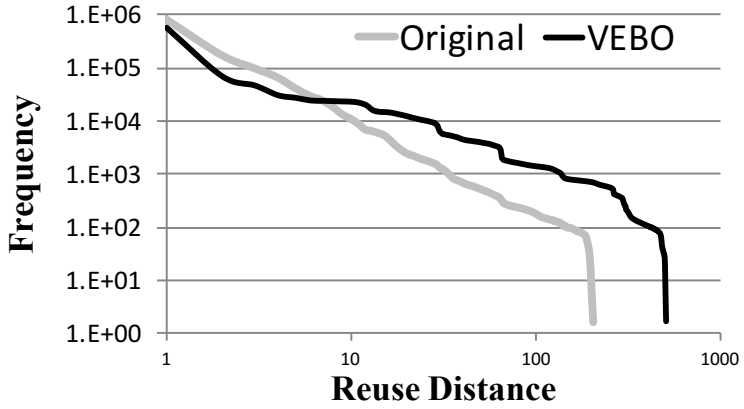


Figure 5.5: Reuse distance distribution of updates to the next frontier in PR for the Powerlaw graph (original version and VEBO reorder version) in GraphGrind using 384 partitions (1st iteration of PR).

check the characteristics of these two graph and find why Gorder loses benefits. Judging on the cache miss rates observed, it appears that the graph was originally formulated in a way that maximises memory locality. Figure 5.5 shows the reuse distance distribution of accesses to the next frontier for the PageRank algorithm applied to the original Powerlaw graph and VEBO Powerlaw graph, as RCM and Gorder version has similar performance as VEBO. Original version has a shorter reuse distances compared to VEBO. It shows that original version has better temporal locality compare reordering graphs. As the graph is a synthetically generated graph (generated by the PBBS generator [3]), this may be a

consequence of the algorithm that generates the graph.

The USARoad graph is not a power-law graph. Rather its degree distribution is close to uniform (the maximum degree is 9). It makes similar observations as for the Powerlaw graph: the cause for performance degradation is a significant degradation of memory locality. Again, the graph is originally ordered in a way that expresses locality well, and it is easier to destroy that locality than it is to improve it.

There is however one curious exception: graph reordering works very well for the Connected Components (CC) algorithm on the USARoad graph. CC uses the label propagation algorithm. Normally, algorithms follow a synchronous approach whereby only data calculated in the previous iteration is propagated. For CC, an asynchronous [59, 95] implementation is correct and results in an *accelerated propagation* of labels: labels determined during one iteration of the algorithm are propagated to other vertices during the same iteration [85]. Graph reordering seems to amplify accelerated propagation, which affects the number of iterations and the density of the frontier (Table 5.8). Gorder and VEBO increase the number of dense iterations compared to the original graph. This can only be explained by accelerating the propagation of values, as this is the only way to make more vertices active sooner. For CC, medium-dense iterations take most of the processing time. Graph reordering significantly reduces the number of medium-dense iterations which has a significant impact on execution time. Many more sparse iterations are required, but these take little time in absolute terms.

Further research is required to understand the interplay between graph reordering and accelerated propagation. It is an open question whether these results are accidental successes or whether graph reordering can be designed to

accelerate asynchronous algorithms.

5.6 Discussion

This chapter has demonstrated that graph processing system using static scheduling benefits strongly from load-balancing. The experiments have considered only shared-memory systems. Distributed systems, however, are typically statically scheduled as migration of the graph topology is expensive. It is interesting future work to validate whether optimisation of the load balance will also improve performance in distributed systems, even if this comes at the expense of a small increase in vertex replication, and thus an increase in the volume of data communication.

Graph processing systems have parameters to affect their operation. For instance, Ligra and Polymer have the direction traversal flag (forward or backward) [85,108]; GraphGrind has a flag to indicate vertex- or edge-orientation [90]; PowerGraph provides alternative scheduling policies [33]. This makes it hard to use these systems as programmers need to carefully deliberate the various options. Using VEBO, however, it can make GraphGrind *parameter-free*, i.e., the system will automatically use the right options. GraphGrind-v1 and GraphGrind-v2 use the vertex- or edge-orientation flag only during loading of the graph, where it decides upon either a vertex-balanced partition or an edge-balanced partition. For graphs pre-processed with VEBO, however, both partitions are equal and the flag has no impact.

5.7 Summary

This chapter has demonstrated that edge-balanced partitioning alone does not create good balancing and that considering vertex-balance along with edge-balance improves load balance significantly. It has designed VEBO, a partitioning algorithm for joint vertex and edge balancing and demonstrated that it achieves optimal load balance for graphs with a power-law degree distribution.

It experimentally evaluated the performance of VEBO on three shared-memory graph processing systems: Ligra, Polymer and GraphGrind. The analysis has shown that the partitioning criteria should be selected in accordance with the graph processing system that will be used. In particular, Polymer and GraphGrind use partitioning to drive static scheduling of parallel loops. As such, load balance is very important for these systems. Ligra, on the other hand, is fully dynamically scheduled. In this case, it is more important to restructure the graph to improve memory locality.

In future work, it would be interesting to investigate whether distributed graph processing systems, which typically use static scheduling, also benefit from increased load balancing and how this trades-off against optimisation of the communication volume. Moreover, it would be valuable to incorporate locality optimisation and edge- and vertex-balance in one algorithm.

While VEBO improves load balance over edge balancing, there remains room for improving the load balance. It will be useful to identify what graph parameters predict load balance, which can lead to better load balance and can also drive the development of new graph analytic algorithms.

Chapter 6

Conclusion and Future Work

6.1 Conclusion.

Graph partitioning is an important technique to efficiently orchestrate the execution of graph analytics. Chapter 3 analyses the performance issues that graph partitioning inadvertently introduces, including load imbalance, increased work per vertex, and a significantly reduced connection density. It classifies characteristics of graph algorithms to *vertex-oriented* and *edge-oriented*, and use this distinction to adapt graph partitioning across NUMA nodes. It particularly shows that graph partitioning incurs an innate performance overhead, which stems from increased control flow and from the decreased connection density of the partitions. Combined, these problems imply that graph partitioning is inherently not scalable to large partition counts.

Chapter 4 differs from prior work by *improving temporal locality* through graph partitioning, which controls the order of iteration over edges. Each partition will be traversed per thread in parallel, and each vertex is updated sequentially per partition. The partitioning algorithm is designed such that each vertex is updated by at most one thread, obviating the need for synchronisation and the use of costly

hardware atomics.

In order to scale to a large number of partitions for better *temporal locality,* this dissertation presents a combination of COO, CSR and CSC graph layouts. This dissertation identifies a third frontier type, *medium-dense* frontiers, which are highly populated but nonetheless benefit from an indexed graph representation such as CSR or CSC. Chapter 4 proposes a fully automated selection of traversal direction based on the density of frontier iteratively. Prior work has not answered in a conclusive manner [8, 85], the reason why a backward or forward traversal order is faster. This dissertation demonstrates that frontier density and memory access order are important factors. GraphGrind achieves significant speedup compared to prior work, out-performing Ligra and Polymer.

Chapter 5 has demonstrated that edge-balanced partitioning alone does not create good balancing and that considering vertex-balance along with edge-balance improves load balance significantly. It designs VEBO, a partitioning algorithm for joint vertex and edge balancing and demonstrated that it achieves optimal load balance for graphs with a power-law degree distribution.

The analysis of performance of VEBO on Ligra, Polymer and GraphGrind has shown that the partitioning criteria should be selected in accordance with the graph processing system that will be used. In particular, Polymer and GraphGrind use partitioning to drive static scheduling of parallel loops. As such, load balance is very important for these systems. Ligra, on the other hand, is fully dynamically scheduled. In this case, it is more important to restructure the graph to improve memory locality.

Above all, this dissertation achieves efficient graph analytics by optimising *load balance*, *NUMA locality and temporal locality.*

6.2 Future work.

Distributed memory systems. All methods of this dissertation are optimising load balance and memory locality of CPUs. Hence, all methods can be applied in distributed memory systems, which is useful to deal with the increasing volume of graph data. It would be valuable to investigate whether distributed graph processing systems, also benefit from increased load balancing and how this trades-off against optimisation of the communication volume.

Graph layouts. There is not a complete understanding of how graph layouts may result in additional speedup of graph analytics. Ligra [85–87] proposes a composite framework with CSR and CSC graph layouts. However, Ligra requires that the programmer simulates both layouts and selects the faster one as target layout. It is an interesting direction to study why different graph layouts have different execution time in graph analytics.

Graph characteristics. Prior work tries to characterise and explain the properties of small-world and power-law graphs through mathematical models of the construction or growth of the graph, such as RMAT [14] model. However, there is not a complete study to investigate whether these key properties relevant to the performance of graph analytics, such as reuse distance distributions to capture memory locality [111], and the replication factor in graph partition [33, 89]. Hence, it is useful to analyse graph analytics by understanding these characteristics of real graphs. It is also an interesting direction to generate synthetic graphs following these characteristics.

Locality improvement of VEBO. VEBO does not optimise locality, it may be lose its benefits for some strong locality graphs, such as USAroad graph

in Chapter 5. Gorder [98] proposes an ordering algorithm to improve CPU cache utilisation. It would be valuable to incorporate locality optimisation and edge- and vertex-balance in one algorithm.

Appendix A

Author's Publications

1. Sun, Jiawen. Student Research Poster: The GraphGrind Framework: Fast Graph Analytics on Large Shared-Memory Systems. Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18). ACM, 2018. Silver Award in Student Research Competition (ACM SRC)

2. Sun, Jiawen, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. " Accelerating Graph Analytics by Utilising the Memory Locality of Graph Partitioning." Proceedings of the International Conference on Parallel Processing (ICPP'17). IEEE, 2017.

3. Sun, Jiawen, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. "GraphGrind: addressing load imbalance of graph partitioning." Proceedings of the International Conference on Supercomputing (ICS'17). ACM, 2017.

4. Sun, Jiawen. Student Research Poster: A Scalable General Purpose System for Large-Scale Graph Processing. Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16). ACM, 2016.

Appendix B

Properties of VEBO

Let $v_{(0)}, v_{(1)}, \ldots, v_{(n-1)}$ be the sequence that holds all vertices in V in order of decreasing in-degree.[1] Assume the algorithm goes through $n = |V|$ steps to place the vertices. Step $t > 0$ corresponds to placing vertex $v_{(t)}$. The algorithm terminates when $t = |V|$.

Definition 1 *Let $w_p(t)$ be the number of edges assigned to partition p before step t, i.e., before placing vertex $v_{(t)}$. The initial situation is $w_p(0) = 0$ for all p.*

Definition 2 *The maximum weight before step t is:*

$$\omega(t) = \max_{j=1,\ldots,P} w_j(t) \tag{B.1}$$

Definition 3 *The minimum weight before step t is:*

$$\mu(t) = \min_{j=1,\ldots,P} w_j(t) \tag{B.2}$$

Definition 4 *The edge imbalance before step t is:*

$$\Delta(t) = \omega(t) - \mu(t) \tag{B.3}$$

An optimal edge placement is achieved when $\Delta(n) \leq 1$, i.e., the number of edges in each partition differs by at most 1. $\Delta(n) = 0$ can occur only when the number of partitions divides into the number of edges.

[1] From here on, we will refer to in-degree as "degree" for brevity.

Definition 5 *The vertex imbalance before step t is:*

$$\delta(t) = \max_{j=1,\ldots,P} u_j(t) - \min_{j=1,\ldots,P} u_j(t) \tag{B.4}$$

where $u_p(t)$ is the number of vertices assigned to partition p before step t.

We first prove the following Lemma that bounds the edge imbalance throughout the placement of vertices. As a short-hand, let $d_{(t)} = deg_{in}(v_{(t)})$ for $t = 0, \ldots, n-1$. Clearly, $d_{(i)} \geq d_{(j)}$ if $i < j$ and $d_{(0)} = N - 1$.

Lemma 1 *When placing a vertex $v_{(t)}$ with degree equal to k when the edge weight is $w_j(t)$ for $j \in 1, \ldots, P$, one of following cases can occur:*

$$\left.\begin{array}{l}\Delta(t+1) \leq \Delta(t) \\ \omega(t+1) = \omega(t)\end{array}\right\} \quad \textit{if} \quad d_{(t)} \leq \Delta(t) \tag{B.5}$$

$$\left.\begin{array}{l}\Delta(t+1) \leq d_{(t)} \\ \omega(t+1) > \omega(t)\end{array}\right\} \quad \textit{if} \quad d_{(t)} > \Delta(t) \tag{B.6}$$

Proof 1 *Algorithm 5 places vertex $v_{(t)}$ on the partition p which has minimal $w_p(t)$, i.e., $w_p(t) = \mu(t)$. The number of edges in partition p is raised to $w_p(t+1) = w_p(t) + d_{(t)}$. By selection of p, $\Delta(t) = \omega(t) - w_p(t)$.*

There are now two cases to consider: (i) if $d_{(t)} \leq \Delta(t)$, then $w_p(t+1) \leq w_p(t) + \Delta(t) = \omega(t)$. As only partition p has a change in weight, this implies that the maximum weight has not been increased by placing $v_{(t)}$ and $\omega(t+1) = \omega(t)$. We moreover know that the minimum weight has not decreased (it will stay the same when there exists another partition with the same weight as p at step t) and thus $\Delta(t+1) \leq \Delta(t)$.

(ii) if $d_{(t)} > \Delta(t)$, then the maximum weight is raised to $\omega(t+1) = w_p(t) + d_{(t)}$ and $\omega(t+1) > \omega(t)$. We now find that

$$\begin{aligned}\Delta(t+1) &= \omega(t+1) - \mu(t+1) \\ &= w_p(t) + d_{(t)} - \mu(t+1) \\ &= \mu(t) + d_{(t)} - \mu(t+1) \\ &\leq d_{(t)}\end{aligned}$$

by observing that the minimum weight cannot decrease from one step to the next.

Intuitively, by placing more edges it either strives towards balancing the edge counts (case (i)), or it is so close to load balance that placing the next vertex must

increase the load imbalance (case (ii)). Importantly, in the latter case, the load imbalance is bounded by the degree of the last vertex placed. As VEBO processes vertices in order of decreasing degree, the edge imbalance reduces throughout the computation.

Now state the edge balance theorem:

Theorem 1 (Edge balance) *Assume a graph $G = (V, E)$ and a number of partitions P. Let $n = |V|$ be the number of vertices and let m be the number of vertices with non-zero degree. Assume that the degree distribution of the graph follows a Zipf distribution with N distinct ranks and scale factor $2 < s < 3$. Let $|E|$ be the number of edges. Assume that P is constrained by $|E| \geq N\,(P-1)$ and that $P < N$. Then, on completion of VEBO (Algorithm 5),*

$$\Delta(n) \leq 1 \tag{B.7}$$

Proof 2 *Our proof builds on the observation that the edge imbalance is bounded each time the maximum weight $\omega(t)$ is increased. If the maximum weight can be increased by placing a degree-1 vertex, then it follows by Lemma 1 that the final edge imbalance is at most 1.*

Assume that the maximum weight is increased after placing vertex $v_{(t)}$ where $0 \leq t < m$, i.e., $\omega(t+1) > \omega(t)$. Assume also that $d_{(t)} > 1$. Without loss of generality, we discard all unplaced vertices with degree equal to $d_{(t)}$ as these will trivially match or exceed $\omega(t+1)$ without increasing load imbalance. Assume $v_{(t')}$ is the first vertex with degree less than $d_{(t)}$. We are interested in the sequence of vertices $v_{(t')}, \ldots, v_{(m-1)}$. As $d_{(t)} > 1$, this set is not empty. Let $E^{'}$ be the set of edges pointing to the vertices $v_{(t')}, \ldots, v_{(m-1)}$. The total number of such edges is $|E^{'}| = \sum_{i=t'}^{m-1} d_{(i)}$. We need to show that $|E^{'}| \geq (P-1)\Delta(t)$, which is the worst-case number of edges that need to be placed to increase the maximum weight. According to Lemma 1, this is satisfied when

$$|E^{'}| \geq (P-1)d_{(t)} \tag{B.8}$$

There are $n\,p_k = n\,k^{-s}/H_{N,s}$ vertices of degree $k-1$ by definition of the degree distribution. Let $k^{'} = d_{(t')}$ be their maximum degree. Then

$$|E^{'}| = \frac{n}{H_{N,s}} \sum_{i=1}^{k^{'}+1} (i-1)\, i^{-s} \tag{B.9}$$

Similarly,

$$|E| = \frac{n}{H_{N,s}} \sum_{i=1}^{N} (i-1) \cdot i^{-s} \tag{B.10}$$

It can be shown that $|E^{'}|/(k^{'}+1) \geq |E|/N$*, which means that the average number of edges per degree decreases as the number of degrees increases in a Zipf distribution. This can be shown using a proof by induction on* N*, which we omit in the interest of brevity. As* $|E|/N \geq (P-1)$*, it follows that* $|E^{'}|/(k^{'}+1) \geq (P-1)$ *and* $|E^{'}| \geq (P-1)(k^{'}+1) \geq (P-1)d_{(t)}$*, which shows that Equation B.8 holds.*

Thus, after increasing the maximum weight at step t *where* $d_{(t)} > 1$*, it can be increased again at some later step. We can repeatedly increase the maximum weight until we reach a step* t *where* $d_{(t)} = 1$*. It follows from Lemma 1 that* $\Delta(t) \leq 1$ *and placement of subsequent vertices cannot increase the edge imbalance, which proves that* $\Delta(m) \leq 1$*.*

Theorem 2 (Vertex balance) *Assume a graph* $G = (V, E)$ *and a number of partitions* P *as required for Theorem 1. Furthermore assume that* $|V| = n > 2\,N$*. Upon placement of vertices* $v_{(0)}, v_{(1)}, \ldots, v_{(m-1)}$ *by Algorithm 5,* $\delta(m) < N/P$ *and upon placement of all vertices by Algorithm 5,* $\delta(n) \leq 1$*.*

Proof 3 *We prove* $\delta(m) < N/P$ *by bounding the vertex load imbalance at each step* t *where* $\omega(t)$ *is increased. Assume that* t_i*,* $i = 0, \ldots, l$ *are the sequence numbers where* $\omega(t_i) > \omega(t_i - 1)$*. For two such sequence numbers* t_{i-1} *and* t_i*, it follows that* $d_{(t_{i-1})} \geq d_{(t_i)}$*. The worst-case vertex load imbalance is* $\delta(t_i) \leq \lceil d_{(t_{i-1})}/d_{(t_i)} - 1 \rceil$ *as one partition is assigned one vertex (namely* $v_{(t_{i-1})}$*), while another may require at most* $\lceil d_{(t_{i-1})}/d_{(t_i)} \rceil$ *vertices of degree* $d_{(t_i)}$ *to reach the same weight. As* $d_{(t_{i-1})}/d_{(t_i)} - 1 = (d_{(t_{i-1})} - d_{(t_i)})/d_{(t_i)} \leq d_{(t_{i-1})} - d_{(t_i)}$*, the total imbalance accumulated is bounded by* $\delta(m) \leq \sum_{i=1}^{l} d_{(t_{i-1})} - d_{(t_i)} = d_{(t_0)} - d_{(t_l)}$*. Note that* $d_{(t_l)} = 1$ *due to Theorem 1 and* $d_{(t_0)} = d_{(0)}$*. A strict upper bound for* $\delta(m)$ *is thus* N*. However, each time the maximum weight is increased, the highest-loaded partition will not be assigned any new vertices until the maximum weight has been increased again. As such, these imbalances are distributed across all partitions and* $\delta(m) < N/P$*.*

Placing now the vertices with zero degree will load balance the vertices in each partition ($\delta(n) \leq 1$*) when there are at least* $\delta(m)\,(P-1)$ *zero-degree vertices. The number of zero-degree vertices is* $n\,p_1 = n/H_{N,s}$*. As* $H_{N,s} < 2$ *for* $s > 1$*, and* $n > 2N$ *by assumption, it follows that* $n/H_{N,s} > n/2 > N$*. We conclude that the number of zero-degree vertices exceeds* N *and therefore also* $\delta(m)\,(P-1) = N\,(P-1)/P$*, which proves that there are sufficient zero-degree vertices available to stack up all partitions with almost the same number of vertices, i.e.,* $\delta(n) \leq 1$*. Note that placing the zero-degree vertices does not affect the previously achieved edge balance.*

This section introduces several constraints in the proofs, namely $P \leq |E|/(N-1)$ and $|V| > 2\,N$. It can be observed from Table 2.2 that these constraints pose no practical limits for actual graphs.

References

[1] Apache software foundation 2012. giraph. `https://giraph.apache.org/` .16,17,18.

[2] A large-scale entity and relation database supporting aggregation of properties. gaffer. `https://github.com/gchq/Gaffer`.

[3] Problem based benchmark suite. pbbs. `http://www.cs.cmu.edu/~pbbs/`.

[4] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[5] Luis A Nunes Amaral, Antonio Scala, Marc Barthelemy, and H Eugene Stanley. Classes of small-world networks. *Proceedings of the national academy of sciences*, 97(21):11149–11152, 2000.

[6] Konstantin Andreev and Harald Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, Nov 2006.

[7] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[8] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[9] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 56–65. IEEE, 2015.

[10] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[11] Florian Bourse, Marc Lelarge, and Milan Vojnovic. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1456–1465. ACM, 2014.

[12] Sergey Brin and Lawrence Page. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer networks*, 56(18):3825–3833, 2012.

[13] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244. ACM, 2009.

[14] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[15] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 444–453, New York, NY, USA, 1999. ACM.

[16] Guojing Cong and Simone Sbaraglia. A study on the locality behavior of minimum spanning tree algorithms. In *High Performance Computing-HiPC 2006*, pages 583–594. Springer, 2006.

[17] Thomas H Cormen. *Introduction to algorithms*. 2009.

[18] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 217–226, New York, NY, USA, 2017. ACM.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Peter J Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005.

[21] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time.

In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 229–241, New York, NY, USA, 1999. ACM.

[22] Chen Ding and Yutao Zhong. Reuse distance analysis. 2001.

[23] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN Notices*, volume 38, pages 245–257. ACM, 2003.

[24] Uriel Feige and Robert Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM Journal on Computing*, 31(4):1090–1118, 2002.

[25] Michael Frasca, Kamesh Madduri, and Padma Raghavan. NUMA-aware graph mining techniques for performance and energy efficiency. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 95:1–95:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[26] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[27] Matteo Frigo, Pablo Halpern, Charles E Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 79–90. ACM, 2009.

[28] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[29] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.

[30] Alan George, Joseph Liu, and Esmond Ng. Computer solution of sparse linear systems. 1994.

[31] Alan George and Joseph W Liu. Computer solution of large sparse positive definite. 1981.

[32] Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. Parallel gibbs sampling: From colored fields to thin junction trees. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 324–332, 2011.

[33] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[34] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of OSDI*, pages 599–613, 2014.

[35] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[36] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proc. VLDB Endow.*, 8(9):950–961, May 2015.

[37] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1:1–1:14, New York, NY, USA, 2014. ACM.

[38] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 77–85. ACM, 2013.

[39] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, volume 7, pages 197–208. Springer, 2007.

[40] Bruce Hendrickson and Jonathan W Berry. Graph analysis with high-performance computing. *Computing in Science & Engineering*, 10(2), 2008.

[41] Bruce Hendrickson and Tamara G Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 26(12):1519–1534, 2000.

[42] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P Sadayappan. Multigraph: Efficient graph processing on gpus. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*, pages 27–40. IEEE, 2017.

[43] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88. IEEE, 2011.

[44] Judith Hurwitz, Alan Nugent, Fern Halper, and Marcia Kaufman. *Big data for dummies*. John Wiley & Sons, 2013.

[45] Louis Ibarra and Dana Richards. Efficient parallel graph algorithms based on open ear decomposition. *Parallel Computing*, 19(8):873–886, 1993.

[46] Intel. *Intel Cilk Plus Language Extension Specification*, version 1.2. 324396-003us edition, September 2013.

[47] Guohua Jin and John Mellor-Crummey. Sfcgen: A framework for efficient generation of multi-dimensional space-filling curves by recursion. *ACM Trans. Math. Softw.*, 31(1):120–148, March 2005.

[48] U Kang, Duen Horng, et al. Inference of beliefs on billion-scale graphs. 2010.

[49] George Karypis. Metis and parmetis. In *Encyclopedia of Parallel Computing*, pages 1117–1124. Springer, 2011.

[50] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–13. IEEE Computer Society, 1998.

[51] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96 – 129, 1998.

[52] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[53] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[54] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, volume 12, pages 31–46, 2012.

[55] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 193–204, New York, NY, USA, 2012. ACM.

[56] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the ab-

sence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[57] L. Li, R. Geda, A. B. Hayes, Y. Chen, P. Chaudhari, E. Z. Zhang, and M. Szegedy. A simple yet effective balanced edge partition model for parallel computing. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1):14:1–14:21, June 2017.

[58] Yongsub Lim, U Kang, and Christos Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3077–3089, 2014.

[59] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[60] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[61] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[62] Michael W Mahoney, LekHeng Lim, and Gunnar E Carlsson. Algorithmic and statistical challenges in modern largescale data analysis are the focus of mmds 2008. *ACM SIGKDD Explorations Newsletter*, 10(2):57–60, 2008.

[63] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[64] Daniel Margo and Margo Seltzer. A scalable distributed graph partitioner. *Proc. VLDB Endow.*, 8(12):1478–1489, August 2015.

[65] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, October 2015.

[66] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[67] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, 2001.

[68] Maged Michael, Jose E Moreira, Doron Shiloach, and Robert W Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[69] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and Analysis of Online Social

Networks. In *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.

[70] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Cache-guided scheduling: Exploiting caches to maximize locality in graph processing.

[71] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[72] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[73] Ana Lucia Varbanescu Niels Doekemeijer. A survey of parallel graph processing frameworks. Technical Report ISSN 1387-2109, Delft Univeristy of Technology Parallel and Distributed Systems Report Series, http://www.pds.ewi.tudelft.nl/fileadmin/pds/reports/2014/PDS-2014-003.pdf, 2014.

[74] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[75] Judea Pearl. *Reverend Bayes on inference engines: A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science, University of California, Los Angeles, 1982.

[76] Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1105–1110. ACM, 2015.

[77] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The tao of parallelism in algorithms. *ACM Sigplan Notices*, 46(6):12–25, 2011.

[78] Michael J Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Surveys (CSUR)*, 16(3):319–348, 1984.

[79] Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM (JACM)*, 55(4):17, 2008.

[80] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[81] Y. Saad. SPARSKIT: A basic tool for sparse matrix computations. Technical Report NASA-CR-185876, NASA, May 1990.

[82] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[83] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[84] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. In *ACM SIGOPS Operating Systems Review*, volume 38, pages 165–176. ACM, 2004.

[85] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[86] Julian Shun. *Shared-memory parallelism can be simple, fast, and scalable*. Morgan & Claypool, 2017.

[87] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *Data Compression Conference (DCC), 2015*, pages 403–412. IEEE, 2015.

[88] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012.

[89] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. Accelerating graph analytics by utilising the memory locality of graph partitioning. In *Parallel Processing (ICPP), 2017 46th International Conference on*, pages 181–190. IEEE, 2017.

[90] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. Graphgrind: addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*, page 16. ACM, 2017.

[91] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[92] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pages 333–342, New York, NY, USA, 2014. ACM.

[93] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[94] Ymir Vigfusson. *Affinity in distributed systems*. PhD thesis, Cornell University, 2010.

[95] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, volume 13, pages 3–6, 2013.

[96] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the Twenty-Second International Conference on Architectural Support*

for Programming Languages and Operating Systems, pages 389–404. ACM, 2017.

[97] Duncan J Watts and Steven H Strogatz. Collective dynamics of small-worldnetworks. *nature*, 393(6684):440–442, 1998.

[98] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828. ACM, 2016.

[99] Wikipedia. Locality of reference, 2017.

[100] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. Sync or async: Time to fuse for distributed graph-parallel computation. *ACM SIGPLAN Notices*, 50(8):194–204, 2015.

[101] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. Distributed power-law graph computing: Theoretical and empirical analysis. In *Advances in Neural Information Processing Systems*, pages 1673–1681, 2014.

[102] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[103] Melissa Yan. Dijkstras algorithm.

[104] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[105] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An experimental comparison of cache-oblivious and cache-conscious

programs. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 93–104, New York, NY, USA, 2007. ACM.

[106] Liang Yuan, Chen Ding, Yunquan Zhang, et al. Modeling the locality in graph traversals. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 138–147. IEEE, 2012.

[107] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[108] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 183–193. ACM, 2015.

[109] Qizhen Zhang, Hongzhi Chen, Da Yan, James Cheng, Boon Thau Loo, and Purushotham Bangalore. Architectural implications on the performance and cost of graph analytics systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 40–51, New York, NY, USA, 2017. ACM.

[110] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Matei Zaharia, and Saman Amarasinghe. Optimizing cache performance for graph analytics. *arXiv preprint arXiv:1608.01362*, 2016.

[111] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):20, 2009.

[112] Jinhong Zhou, Chongchong Xu, Xianglan Chen, Chao Wang, and Xuehai Zhou. Mermaid: Integrating vertex-centric with edge-centric for real-world graph processing. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 780–783. IEEE Press, 2017.